

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！



大话 Java性能优化

周明耀 / 著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

周明耀 ▶



- 12年投资银行项目、分布式计算项目工作经验，IBM开发者论坛专家作者。
- 一名IT技术狂热爱好者，一名顽强到底的工程师。推崇技术创新、思维创新，对于新技术非常热爱，致力于技术研发、研究，通过发布文章、书籍、互动活动的形式积极推广软件技术。
- 欢迎添加微信“michael_tec”，共同探讨IT技术话题。

内容简介

大话 Java性能优化

周明耀 / 著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书主要提供 Java 性能调优方面的参考建议及经验交流。作者力求做到知识的综合传播,而不是仅仅只针对 Java 虚拟机调优进行讲解,另外力求每一章节都有实际的案例支撑。具体包括:性能优化策略、程序编写及硬件服务器的基础知识、Java API 优化建议、算法类程序的优化建议、并行计算优化建议、Java 程序性能监控及检测、JVM 原理知识、其他相关优化知识等。

通读本书后,读者可以深入了解 Java 性能调优的许多主题及相关的综合性知识。读者也可以把本书作为参考,对于感兴趣的主题,直接跳到相应章节寻找答案。

总的来说,性能调优在很大程度上是一门艺术,解决的 Java 性能问题越多,技艺才会越精湛。我们不仅要关心 JVM 的持续演进,也要积极地去了解底层的硬件平台和操作系统的进步。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

大话 Java 性能优化 / 周明耀著. —北京: 电子工业出版社, 2016.4
ISBN 978-7-121-28481-6

I. ①大… II. ①周… III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字 (2016) 第 063438 号

责任编辑: 董 英

印 刷: 北京京科印刷有限公司

装 订: 三河市皇庄路通装订厂

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱

邮编: 100036

开 本: 787×1092 1/16

印张: 35.25

字数: 993 千字

版 次: 2016 年 4 月第 1 版

印 次: 2016 年 4 月第 1 次印刷

印 数: 3000 册 定价: 89.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zltts@phei.com.cn, 盗版侵权举报请发邮件到 dbqq@phei.com.cn。

服务热线: (010) 88258888。

序 言

最大的思想紊乱是相信人们想要相信的事情。

——路易斯·巴斯德 (Louis Pasteur)

Michael 周是个具有丰富程序经验的架构师和项目管理者，他从国内作坊式的软件开发公司起步，经历了著名的咨询公司凯捷的欧洲工作洗礼，后来于美国花旗软件担任高级软件技术总监，平时常常思考和总结 21 世纪以来我国软件开发者的，特别是 Java 开发工程师的困惑。

我们通常情况下，一开始可以有有条不紊地进行软件需求定义和分析，随着上线时间的不断逼近，面对客户的咄咄逼人的需求修改和即刻变更需求上线压力，程序员作为弱势群体，往往会考虑时间优先原则，很难守住按部就班的开发计划和开发方式，从而导致出现了软件质量的大幅下降。软件一定存在修改的余地，但是程序员们通常不相信自己的系统存在诸多问题，尤其是感觉自己已经做得相当完美。系统调优在软件的后续改进和重构中占有很大的地位，能够弥补前述的不足，本书以通俗的语言和引人入胜的故事，重点讲述软件性能调优的方法论和具体实现路径，读者可以根据自己的实际情况进行参照比对，就像进了兵器库挑选合适自己的顺手武器。

程序凑合着上线是一回事，而能够优美地运行在压力下往往很不容易。本书对于所有有志于进行软件高级管理的人员而言，具有非常重要的意义。

海适云承 CEO 兼首席架构师 沈英桓 (Sam Shen)

前言

7 岁那年，当我合上《上下五千年》一套三册书籍时，我对自己说，我想当个作家。这一晃 27 年了，等待了 27 年，我的第一本书《大话 Java 性能优化》即将面世了。我是多么的忐忑、惊喜，就像第一次面对我的女儿“小顽子”，给她取这个小名，希望她顽强到底，因为我相信，你若顽强到底，一切皆有可能。

从 15 岁拥有自己第一台电脑算起，已经有接近 20 年的计算机学习时间，加上 11 年的工作经历，我对于工作，对于工程师这个职业，有一些自己的感悟。我认为，职业素养非常重要。

1929 年，在汪精卫的支持下，余云岫等人提出了全面废除中医、禁止中医的提案，并很快获得初审通过。在这样的局面下，全国各地中医师多次到南京请愿，虽有孙科等人的支持，但反响不大。相持阶段，无独有偶，汪精卫的岳母身患痢疾，西医师医治无效，京城四大名医之一的施今墨先生毅然赴汪府。施今墨凭脉，每言必中，使汪精卫的岳母心服口服，频频点头称是。处方时施今墨说：“安心服药，一诊可愈，不必复诊。”病危至此，一诊可愈？众人皆疑。据此处方仅服数剂，果如施今墨所言。汪精卫不得不服中医，最终撤回提案。施老先生医德高尚，死后遗体都捐献出来供科学研究，绝不是阿谀奉承之人，他赴汪府，完全是因为对中医生这个职业的尊重，为了让人知道中医的深奥。

戒口

佛教五戒之一的不妄语，要求我们不欺骗他人、不在不清楚实际情况的时候胡乱说话，放到职场，也可以加上信息安全的要求。

《越绝书》载文种述九术时说：“故曰九者勿患，戒口勿传，以取天下不难，况于吴乎？”文种希望勾践秘而不宣，以免人多口杂，泄露机密。每个人都有自己的岗位、职责，我们要做的是做好自己的事情，不对不属于自己工作范围内的事情评价、传播，不在背后说同事的坏话。作为一名技术人员，如果不能做到戒口、静心、专心，那我觉得你应该尽早转行，你不适合，也绝不会成为一名技术大拿。

气场

一位职业的工作者，他身上有一种称为气场的东西存在。人的气场是看不见的，但这种力量是巨大的，就像万有引力一样，我们每个人身上的这种气场无时无刻不在影响你的人生。这种气场的行程与你的观念、信仰、环境、朋友、呼吸、事物、欲望、静息与睡眠相关。一个人的气质

很好，外表精神、有修养、有道德，这个人的气场就好，就会吸引好的事，吸引好的运气。每个人都会遇到各种各样的苦难，但是我坚信，你若顽强到底，一切皆有可能。

教养

看不见的教养很难。在乌合之众中谁能保持优雅和教养？在群体无意识中谁能保持清醒和判断？更难的是那些“慎独”的教养。日本有一种文化，叫作“不给别人添麻烦”的文化，我们每个人在做事之前都应该考虑是否自己的行为会给别人造成麻烦。教养不是道德规范，也不是小学生行为准则，其实也并不跟文化程度、社会发展、经济水平挂钩，它更是一种体谅，体谅别人的不容易，体谅别人的处境和习惯。对于教养，我个人的理解是，谦逊是一种教养，自尊更是。

心态

尼克·胡哲说过，人们经常埋怨什么也做不来，但如果我们只记挂着想拥有或欠缺的东西，而不去珍惜所拥有的，那根本改变不了问题！真正改变命运的，并不是我们的机遇，而是我们的态度。

一个人的心态很是重要，心量小的人，芝麻大小的事情也能在心里翻江倒海。心量大的人，即使在危机面前也能镇静自若。同样一件事情，掀起的波澜大小却因人而异。有一句话很好，用于技术人员我觉得尤其合适，“想要成为一棵大树，就不要去和草争”。

一个人的成就，不得以金钱衡量，而是一生中，你善待过多少人，有多少人怀念你。成功并非单指事业，无论是爱好或职业上的成功都只是成就。成功应该是多元化的，如人的一生包含了很多追求一样，而非单一指向。然后，无论你多有成就，真正的成功，就是陪伴家人。所有的情感都是需要陪伴的，这些陪伴成为一个个美好的回忆，这些都是整个家庭最宝贵、最重要的财富，这些远远超越物质的重要性。在中国，因为价值观相对比较单一，社会显得很浮躁、很物质，所以大多以物质的追求为主，越多越好，内心也想过美好的生活。但当你的心完全趋向金钱的时候，很多美好的东西就会自动屏蔽了，不会出现在生活中。别让忙碌空白了回忆。

此外，作为一名技术人员，我觉得，职业生涯中可能很多次需要面对工作的变换、角色的变化，有很多知识需要学习，所以，我们应该把“归零”当成一种生活的新常态。

劝学

我觉得有一句话总结得特别好，“能干工作、干好工作职场生存的基本保障”。

荀子是儒家八派中一派的创始人，其思想学说以儒家为本，兼采道、法、名、墨诸家之长。荀子在他的著作《劝学》一文中这样写道，“君子曰：学不可以已。青，取之于蓝，而青于蓝；冰，水为之，而寒于水。”这段文字大体表达了学习是不可以停止的，君子广泛学习并且每天反省自己，就会明白道理，行为上也不会有什么过错。

全球成功的科技型企业，无论是微软的比尔·盖茨，还是苹果的乔布斯，Facebook 的扎克伯

格，无一不是技术专家，创新型企业必须由这样的企业家带队，懂技术，就会站在前沿。对于大型科技企业而言，光懂技术不够，还要懂市场。

诸葛亮在给他的儿子写的著名的《诫子书》中指出，宁静才能够修养身心，静思反省。不能够静下来，则不可以有效地计划未来，而且学习的首要条件，就是有宁静的环境。审慎理财，量入为出，不但可以摆脱负债的困扰，更可以过着简朴的生活，不会成为物质的奴隶。要计划人生，不要事事讲求名利，才能够了解自己的志向，要静下来，才能够细心计划将来。学习需要专注，平静心境才能事半功倍。学习的过程中，决心和毅力非常重要，因为缺乏了意志力，就会半途而废。拖延就不能够快速地掌握要点。时光飞逝，意志力也会随着时间消磨。

归属感

每个足球队有 11 位球员在球场上比赛，估计最不引人注目的应该是守门员了吧，他要忍受着大多数时间的无聊，还要保持着警惕。当危机发生时，很有可能还要一个人战斗，需要勇敢地面对对方前锋，唯一的目标是，绝对不让你攻破球门。我们很多时候可能也是如此，艰苦奋斗，当解决了某个问题，或是帮助公司拿到某个招标，我们都会感到自豪感、成就感，这就是归属感，对于技术领域的归属感。

最后，自我介绍一下，我叫周明耀，研究生学历，一名九三学社社员，12 年工作经验，IBM 开发者论坛专家作者。我是一名 IT 技术狂热爱好者，一名顽强到底的工程师。我推崇技术创新、思维创新，对于新技术非常热爱。

感谢我的家人，和谐的家庭帮助我完成了这本书，我的妻子，她美丽、细心、博学、偶尔不那么温柔，但是我很爱她。我的小顽子，她天生的性格很像我，希望她能够踏踏实实做人，保持创新精神，平平安安、健健康康地生活下去。感谢我妻子父母、我的父母，他们帮我照顾小孩，我才有时间编写此书。感谢浙江省特级教师、杭州高级化学老师郑克良老师，郑老师的一句“永远不要放弃”，推动着我多年的发展。感谢数学老师张老师在公开场合对我智商的褒奖，第一次收获这样的赞赏，对我这样性格的孩子是多么的重要，谢谢。感谢王芳同学，因为你的插画天赋，让这本书的内容更加丰富、可读，不要忽视了自己的才华，你很有天赋。

我相信这本书不是终点，它是麦克叔叔此生一系列技术书籍的开端，下一本书籍见。

目 录

第1章 性能调优策略概述	1
1.1 为什么需要调优	1
1.2 性能优化的参考因素	5
1.2.1 传统计算机体系的分歧	5
1.2.2 导致系统瓶颈的计算资源	7
1.2.3 程序性能衡量指标	8
1.2.4 性能优化目标	9
1.2.5 性能优化策略	10
1.3 性能调优分类方法	11
1.3.1 业务方面	12
1.3.2 基础技术方面	12
1.3.3 组件方面	17
1.3.4 架构方面	19
1.3.5 层次方面	20
1.4 本章小结	21
第2章 优化前的准备知识	22
2.1 服务器知识	23
2.1.1 内存	23
2.1.2 GPU/CPU	44
2.1.3 硬盘	49
2.1.4 网络架构	51
2.2 新兴技术	53
第3章 Java API 调用优化建议	54
3.1 面向对象及基础类型	55
3.1.1 采用 Clone()方式创建对象	55
3.1.2 避免对 boolean 判断	55
3.1.3 多用条件操作符	56

3.1.4	静态方法代替实例方法.....	56
3.1.5	有条件地使用 final 关键字	58
3.1.6	避免不需要的 instanceof 操作	58
3.1.7	避免子类中存在父类转换.....	59
3.1.8	建议多使用局部变量.....	60
3.1.9	运算效率最高的方式——位运算.....	60
3.1.10	用一维数组代替二维数组.....	62
3.1.11	布尔运算代替位运算.....	64
3.1.12	提取表达式优化.....	65
3.1.13	不要总是使用取反操作符(!).....	66
3.1.14	不要重复初始化变量.....	66
3.1.15	变量初始化过程思考.....	66
3.1.16	对象的创建、访问过程.....	69
3.1.17	在 switch 语句中使用字符串	70
3.1.18	数值字面量的改进.....	73
3.1.19	优化变长参数的方法调用.....	74
3.1.20	针对基本数据类型的优化.....	75
3.1.21	空变量	76
3.2	集合类概念.....	77
3.2.1	快速删除 List 里面的数据	78
3.2.2	集合内部避免返回 null	80
3.2.3	ArrayList、LinkedList 比较.....	82
3.2.4	Vector、HashTable 比较	85
3.2.5	HashMap 使用经验	87
3.2.6	EnumSet、EnumMap	91
3.2.7	HashSet 使用经验	92
3.2.8	LinkedHashMap、TreeMap 比较	96
3.2.9	集合处理优化新方案.....	99
3.2.10	优先考虑并行计算.....	107
3.3	字符串概念.....	108
3.3.1	String 对象.....	108
3.3.2	善用 String 对象的 SubString 方法	111
3.3.3	用 charat()代替 startswith()	113
3.3.4	在字符串相加的时候,使用'代替" "	114
3.3.5	字符串切割	114
3.3.6	字符串重编码	117
3.3.7	合并字符串	118
3.3.8	正则表达式不是万能的.....	122
3.4	引用类型概念.....	123

3.4.1	强引用 (Strong Reference)	126
3.4.2	软引用 (Soft Reference)	131
3.4.3	弱引用 (Weak Reference)	135
3.4.4	引用队列	141
3.4.5	虚引用 (Phantom Reference)	142
3.5	其他相关概念	146
3.5.1	JNI 技术提升	146
3.5.2	异常捕获机制	150
3.5.3	ExceptionUtils 类	154
3.5.4	循环技巧	155
3.5.5	替换 switch	157
3.5.6	优化循环	158
3.5.7	使用 arrayCopy()	159
3.5.8	使用 Buffer 进行 I/O 操作	161
3.5.9	使用 clone()代替 new	164
3.5.10	I/O 速度	166
3.5.11	Finally 方法里面释放或者关闭资源占用	167
3.5.12	资源管理机制	167
3.5.13	牺牲 CPU 时间	169
3.5.14	对象操作	172
3.5.15	正则表达式	172
3.5.16	压缩文件处理	174
3.6	本章小结	175
第 4 章	程序设计优化建议	176
4.1	算法优化概述	176
4.1.1	常用算法逻辑描述	177
4.1.2	多核算法优化原理	186
4.1.3	Java 算法优化实践	188
4.2	设计模式	196
4.2.1	设计模式的六大准则	196
4.2.2	单一对象控制	200
4.2.3	并行程序设计模式	202
4.2.4	接口适配	205
4.2.5	访问方式隔离	219
4.3	I/O 及网络相关优化	225
4.3.1	I/O 操作优化	225
4.3.2	Socket 编程	231
4.3.3	NIO 2.0 文件系统	235

4.4 数据应用优化.....	236
4.4.1 关系型数据库优化.....	236
4.4.2 向 HBase 插入大量数据.....	240
4.4.3 解决海量数据缓存.....	251
4.5 其他优化.....	256
4.5.1 Web 系统性能优化建议.....	256
4.5.2 死锁情况解决方案.....	259
4.5.3 JavaBeans 组件.....	268
4.6 本章小结.....	269
第 5 章 Java 并程序序优化建议.....	270
5.1 并程序序优化概述.....	270
5.1.1 资源限制带来的挑战.....	271
5.1.2 进程、线程、协程.....	272
5.1.3 使用多线程的原因.....	281
5.1.4 线程不安全范例.....	282
5.1.5 重排序机制.....	284
5.1.6 实例变量的数据共享.....	286
5.1.7 生产者与消费者模式.....	288
5.1.8 线程池的使用.....	290
5.2 锁机制对比.....	296
5.2.1 锁机制概述.....	296
5.2.2 Synchronized 使用技巧.....	298
5.2.3 Volatile 的使用技巧.....	303
5.2.4 队列同步器.....	304
5.2.5 可重入锁.....	307
5.2.6 读写锁.....	308
5.2.7 偏向锁和轻量级锁.....	309
5.3 增加程序并行性.....	310
5.3.1 并发计数器.....	311
5.3.2 减少上下文切换次数.....	312
5.3.3 针对 Thread 类的更新.....	314
5.3.4 Fork/Join 框架.....	314
5.3.5 Executor 框架.....	318
5.4 JDK 类库使用.....	319
5.4.1 原子值.....	320
5.4.2 并行容器.....	324
5.4.3 非阻塞队列.....	332
5.4.4 阻塞队列.....	338

5.4.5 并发工具类	365
5.5 本章小结	376
第6章 JVM 性能测试及监控	377
6.1 监控计算机设备层	378
6.1.1 监控 CPU	380
6.1.2 监控内存	405
6.1.3 监控磁盘	417
6.1.4 监控网络	423
6.2 监控 JVM 活动	428
6.2.1 监控垃圾收集目的	429
6.2.2 GC 垃圾回收报告分析	430
6.2.3 图形化工具	431
6.2.4 GC 跟踪示例	437
6.3 本章小结	438
第7章 JVM 性能调优建议	439
7.1 JVM 相关概念	439
7.1.1 内存使用相关概念	440
7.1.2 字节码相关知识	443
7.1.3 自动内存管理	448
7.2 JVM 系统架构	451
7.2.1 JVM 的基本架构	451
7.2.2 JVM 初始化过程	453
7.2.3 JVM 架构模型与执行引擎	456
7.2.4 解释器与 JIT 编译器	456
7.2.5 类加载机制	457
7.2.6 虚拟机	458
7.3 垃圾回收机制相关	459
7.3.1 GC 相关概念	459
7.3.2 垃圾回收算法	468
7.3.3 垃圾收集器	476
7.4 实用 JVM 实验	490
7.4.1 将新对象预留在年轻代	490
7.4.2 大对象进入年老代	494
7.4.3 设置对象进入年老代的年龄	495
7.4.4 稳定与震荡的堆大小	497
7.4.5 吞吐量优先案例	498

7.4.6	使用大页案例	499
7.4.7	降低停顿案例	499
7.4.8	设置最大堆内存	499
7.4.9	设置最小堆内存	500
7.4.10	设置年轻代	503
7.4.11	设置持久代	504
7.4.12	设置线程栈	504
7.4.13	堆的比例分配	505
7.4.14	堆分配参数总结	508
7.4.15	垃圾回收器相关参数总结	509
7.4.16	查询 GC 命令	515
7.5	本章小结	515
第 8 章	其他优化建议	516
8.1	Java 现有机制及未来发展	516
8.1.1	Java 体系结构变化历史	516
8.1.2	Java 语言面临的挑战	520
8.1.3	Java 8 的新特性	522
8.1.4	Java 语言前景	523
8.1.5	物联网: Java 和你是一对	524
8.1.6	Java 模块化发展	525
8.1.7	OpenJDK 的发展	527
8.2	系统架构优化建议	528
8.2.1	系统架构调优	528
8.2.2	Java 项目优化方式分享	530
8.2.3	面向服务架构	534
8.2.4	程序隔离技术	538
8.2.5	团队并行开发准则	544
8.3	与编程无关	546
8.3.1	工程师品格	546
8.3.2	如何成为技术大牛	547
8.3.3	编程方法分享	548
8.4	本章小结	549

1 chapter

第1章 性能调优策略概述

2011年1月，新加坡飞往杭州的航班。飞行持续时间很长，大约6个小时，坐在四周的人很快熟悉了，互相攀谈起来。有一位小姑娘，十六七岁的模样，长得很漂亮，默默地坐在座位上。热心的阿姨和她攀谈，问起她的情况，她带着疲倦自我介绍起来，“我在新加坡念初三，那所学校一点都不好，我在成都是最好的初中毕业的，也考上了成都最好的高中，但是，我的父母，他们一定要我来新加坡复读初三，让我考新加坡的高中，我一点都不喜欢这里，这里的同学看不起我们这些大陆学生，经常上课找大陆来的老师麻烦，经常辱骂我们，我烦透了!!!”对，这不是自我介绍，这是一个人接近奔溃边缘的歇斯底里。也就是在当时，我做出了决定，我绝不会让我的女儿这样远离我，一个人在很年幼的时候就必须独立面对生活的困难，绝不。无论她的父母出于什么原因让她去国外念书，我所看到的，是让一个不适合承受压力的人承担了巨大的压力，这就是本书的编写原因。在这本书里我想要和大家讨论的话题是基于Java语言的性能优化，我们不能随意地给出性能优化方案，就像随意指派由那位小姑娘来完成全家的未来方向一样。我们必须经过严密的研究、测试及验证，明确造成性能瓶颈真正的原因后才能开始着手，盲目地行动只会造成不必要的损失。当然，如果系统架构设计得很好，就可以在很大程度上避免类似事情发生，这不是本书的主要讨论范围。

本章主要介绍和解决以下问题，这些也是全书的基础：

- 为什么需要调优，这是您阅读本书的依据，只为需要调优而调优。
- 了解程序性能的各项指标，包括物理机器性能、程序性能。
- 性能调优分类方法，包括调优方向、调优方法、调优层次。

1.1 为什么需要调优

注意，这一节会提到许多技术名词，本着让Java初学者看懂本书的目的，笔者尽量第一时间做出注释，如有遗漏读者可以阅读后续章节，均有详细介绍。

经历了多年的发展, Java 已由一门单纯的计算机编程语言, 逐渐演变为一套强大的技术体系平台。根据不同的技术规范, Java 设计者们将 Java 划分为 3 种结构独立但却又彼此依赖的技术体系分支, 分别是 Java SE、Java EE 和 Java ME¹, 其中 Java EE 被广泛使用在企业级领域, 除了包括 Java API 组件外, 还扩充有 Web 组件、事务组件、分布式组件、EJB 组件、消息组件等, 并持续发展到现在。综合 Java EE 的这些技术, 开发人员可以构建出一个具备高性能、结构严谨的企业级应用, 并且 Java EE 也是用于构建 SOA 架构的首选平台。

Java 的持续发展要感谢 Google, 正是 Google 将 Java 作为 Android 操作系统的应用层编程语言, 使得 Java 可以在 PC 时代、移动互联网时代都得到迅猛发展, 可以用于手持移动设备、嵌入式设备、个人电脑、高性能的集群服务器或大型机。

随着互联网业务的不断拓展、繁荣, 越来越多的系统架构开始参照互联网企业的系统架构方式。无论是互联网、物联网, 还是传统行业的软件设计, 笔者认为, 任何技术都离不开对业务需求的支撑², 所以开始展开研究程序性能问题之前, 我们需要先了解系统业务逻辑。

铁道部的 12306³网站一直被全国人民所诟病, 它确实存在一些问题, 但是这些看似简单的问题, 其背后牵扯着复杂的系统架构设计。这些设计最终是为业务需求服务的, 即 12306 的职责是为所有旅客的需求服务的, 而程序员设计的程序又是为 12306 服务的, 所有的用户体验归到最终就是服务意识。我们来看一下 12306 的业务, 12306 需要支持海量并发查询, 即海量用户同时查时间、查车次、查座位、查铺位。此外, 对应的下单过程也就会伴随着海量并发的数据库操作。据说, 淘宝在双十一期间也只有几百万用户⁴, 而春运期间抢购火车票是全国人民的统一活动, 瞬时访问数量有千万级别甚至是亿级别的。据说 12306 的高峰访问是 10 亿 PV⁵, 这些访问主要集中在早 8 点到 10 点, 每秒 PV 在高峰时上千万⁶。

再来看看其他的业务系统。奥运会期间的奥运票务系统采用抽奖的方式, 这样的业务设计让系统不存在先来先得抢购需求, 由于是事后抽奖, 因此事前只负责收集信息, 所以不需要保证数据的一致性, 这也就没有高强度并发锁⁷的需求, 很容易通过水平扩展方式克服性能瓶颈。B2C 网站一般实时性要求不高, 比如下单, 用户提交订单后, 订单并不是马上被处理的, 而是等待一定时间后, 用户才会收到订单是否确认的通知, 这样就确保了数据不需要立即被处理, 没有了数据高并发同步的需求。也就是说, 在高并发要求下的数据一致性是通常情况下的性能瓶颈点, 也是通常意义上的技术难点之一。

前面提起过, 高并发情况下的数据高度实时一致性需求是很难实现的。对于一个网站来说, 并发浏览网页造成的高负载较容易处理, 高并发的查询负载也可以处理, 但是实时下单是最难处

¹ Java SE (Java Platform, Standard Edition); Java EE (Java Platform, Enterprise Edition); Java ME (Java Platform, Micro Edition)。

² 例如金融系统, 不能单纯按照程序员的思维设计系统和数据库结构, 而是应该更加紧密地与业务结合。

³ 12306: 中国铁路客户服务中心 (12306 网) 是铁路服务客户的重要窗口, 将集成全路客货运输信息, 为社会和铁路客户提供客货运输业务和公共信息查询服务。

⁴ 来源 2013 年的网络数据, 作者非阿里人士, 也没有淘宝账户, 所以数据不准确请读者见谅。

⁵ PV: 即页面浏览量, 通常是衡量一个网络新闻频道或网站甚至一条网络新闻的主要指标。

⁶ 摘自 2013 年的官方数据。

⁷ 提高系统并发吞吐能力的方式, 第 5 章会详细介绍。

理的,因为下单需要访问当前的库存量,对于12306网站来说,库存量就是指火车票的库存,由于这是一个全国联网系统,所以可以预见库存量保持数据一致性的难度。据说苹果CEO库克⁸正是因为处理好了库存问题才得以继任乔帮主⁹的宝座。目前来看,很多B2C¹⁰网站的下单都是通过异步方式来实现的,这样的做法可以避免数据高度一致性要求。

淘宝模式相较于传统B2C网站有一个优势,即它不需要查询库存。B2C网站拥有自己的仓库,每次下单前,都需要查找距离客户最近的仓库是否有库存,这样的计算量累计后会很大。比如,你在上海买一本书,如果上海附近的仓库没货,我们需要先计算哪个仓库既离上海最近又有这本书。淘宝网站由于本身商业模式的原因,它不需要去实时检查库存,反而对于性能扩展较为容易。

的确我们可以通过Nginx¹¹来搞定每秒10万的静态请求,只要有足够的网络带宽、磁盘I/O,服务器的并发计算能力够强,可以很容易地处理10万的并发连接。但是如果引入了大量的业务逻辑,那就不是单纯的访问问题了,该解决方案也就成了浮云。

除了业务需求、程序运行方式之外,程序设计本身需要考虑基础编程技术、系统架构、网络技术、操作系统、硬件服务器等诸多因素。计算机专家在问题求解时非常重视表达式简洁性的价值。UNIX的先驱者Ken Thompson¹²曾经说过非常著名的一句话:“丢弃1000行代码的那一天是我最有成效的一天之一。”这对于任何一个需要持续支持和维护的软件项目来说,都是一个当之无愧的目标。早期的Lisp¹³贡献者Paul Graham¹⁴甚至将语言的简洁性等同为语言的能力。这种对能力的认识让我们把编写紧凑、简洁的代码作为许多现代软件项目选择语言的首要标准。

任何程序都可以通过重构代码方式去除多余的代码或无用的占位符,例如空格,删除空格后会让代码变得更加简短。不过某些语言天生就善于表达,也就特别适合于简短程序的编写。APL语言的设计理念是利用特殊的图形符号让程序员用很少量的代码就可以编写功能强大的程序。这类程序如果实现得当,可以很好地映射成标准的数学表达式。简洁的语言在快速创建小脚本时非常高效,特别是在目的不会被简洁所掩盖的简洁明确的问题域中。

相比于其他程序设计语言,Java语言的冗长已经名声在外,主要原因是由于程序开发社区中所形成的惯例。在完成任务时,很多情况下要更大程度地考虑描述性和控制能力。例如,长期来看,长变量名会让大型代码库的可读性和可维护性更强。描述性的类名通常会映射为文件名,在向已有系统中增加新功能时会显得很清晰。如果能够一直坚持下去,描述性名称可以极大简化用于表明应用中某一特定的功能的文本搜索。实践证明,这些定义方式让Java在大型复杂代码库的

⁸ 蒂姆·库克(Tim Cook),1960年11月1日出生于美国阿拉巴马州,1982年毕业于奥本大学工业工程专业。1988年获得杜克大学企业管理硕士学位。曾在IBM供职12年,负责PC部门在北美和拉美的制造和分销运作。1998年年初,库克进入苹果,任副总裁,主管苹果的电脑制造业务。2011年接替乔布斯担任苹果公司CEO。

⁹ 即乔布斯,1955年02月24日—2011年10月05日。

¹⁰ B2C: Business To Customer。

¹¹ 一个高性能的HTTP和反向代理服务器,也是一个IMAP/POP3/SMTP服务器。

¹² 肯·汤普森(Kenneth Lane Thompson,1943年2月4日),一般称之为Ken Thompson,为美国计算机科学学者,与丹尼斯·里奇同为1983年图灵奖得主。

¹³ LISP是一种通用高级计算机程序语言,长期以来垄断人工智能领域的应用。Lisp作为因应人工智能而设计的语言,是第一个函数式程序设计语言,有别于C、Fortran等命令式程序设计语言和Java、C#等面向对象语言。

¹⁴ 保罗·格雷厄姆(Paul Graham),美国著名程序员、风险投资家、博客和技术作家。他以Lisp方面的工作而知名,也是最早的Web应用Viaweb的创办者之一,后来以近5千万美元的价格被雅虎收购,成为Yahoo! Store。

大规模实现中取得了极大的成功。

相对于传统的 32 位虚拟机, 64 位虚拟机所具备的最大优势就是可以访问大内存。32 位虚拟机最大可用内存空间被限定在了 4GB, 并且 Java 堆区的大小配置存在最大限制, 如果是在 Windows 平台下最大只能设置到 1.5GB, 而在 Linux 平台下最大也只能设置到 2GB~3GB。也就是说, Java 堆区的内存大小设置还需要依赖于具体的操作系统平台。

既然 32 位虚拟机无法满足大内存消耗的应用场景, 那么 64 位虚拟机的出现则是顺理成章的。64 位虚拟机之所以能够访问大内存, 是因为其采用了 64 位的指针架构, 这也是寻址访问大内存的关键要素。

在 JDK1.6 Update14 版本之前, 64 位虚拟机的综合性能表现实际上是不如 32 位虚拟机的, 这主要是因为 OOPS (Ordinary Object Pointers, 普通对象指针) 从 32 位膨胀到 64 位后, CPU Cache Line 中的可用 OOPS 变少, 这样一来就会直接影响并降低 CPU 的缓存使用率, 这就是 64 位虚拟机在性能上之所以落后于 32 位虚拟机的主要原因。其次, 由于部署在 64 位虚拟机上的性能都需要用到大内存, 尤其是互联网项目, 经常需要使用多达几十乃至几百 GB 的内存, 这对于传统的 32 位虚拟机将无法承载, 只能依靠 64 位虚拟机去支撑。但是管理这么大的内存开销对于 GC (Garbage Collection) 来说将会是一场非常严峻的考验, 甚至很有可能会导致 GC 在执行内存回收期间消耗更长的时间, 同时也意味着工作线程的等待时间将会延长。如今随着 64 位虚拟机的逐渐成熟, 指针压缩将会通过对齐补白等操作将 64 位指针压缩为 32 位, 以此改善 CPU 缓存使用率达到提升 64 位虚拟机运行性能的目的。

对于小型项目来说, 简洁性则更受青睐, 某些语言非常适于短脚本编写或者在命令提示符下的交互式探索编程。Java 作为通用性语言, 则更适用于编写跨平台的工具。在这种情况下, “冗长 Java” 的使用并不一定能够带来额外的价值。虽然在变量命名等方面, 代码风格可以改变, 不过从历史情况来看, 在一些基本的层面上, 与其他语言相比, 完成同样的任务, Java 语言仍需更多的字符。为了应对这些限制, Java 语言一直在不断地更新, 尝试包含一些通常称为“语法糖”的功能。用这些习语可以实现用更少的字符表示相同功能的目标, 与其对应的更加冗长的配对物相比, 这些习语更受程序开发社区的欢迎, 也会被社区作为通用用法快速地采用。

现代 CPU 架构将多核、多硬件执行线程技术推向前台, 这意味着我们可以利用更多的 CPU 资源做更多的工作。然而, 要利用好这些额外的 CPU 资源, 运行于其上的程序必须要能够支持并行工作这个需求。通俗点讲, 这些程序需要按照多线程的方式构造或设计才能充分地利用额外的硬件线程。

最近这几年, 服务器端网络使用的基础通信技术并没有取得太大的进步。服务器端大多数在绝不允许服务中断的关键任务环境中, 新技术很难渗透, 也很难植根于这样的环境。但正因如此, 服务器端的多余部分才得以被剔除, 逐渐地形成了非常精简单纯的风格。网络的基础技术可以说已经成型了, 然而在网络上运行的网络设备和服务器的技术仍然踩着现在进行时的节奏在持续不断地爆发性发展, 由此出现了虚拟技术和网络存储技术等基于网络的创新技术。如今, 它们已经在系统中不可或缺。随着这些技术的发展, 人们追求的网络形态和网络设计的方式也在时刻发生着变化, 基础架构工程师和服务器工程师必须能灵活应对这些变化才行。对应地, 软件设计程序员也需要有针对性地做出应对措施。

虚拟化技术是一种资源管理技术，是将计算机的各种实体资源，如服务器、网络、内存及存储等，予以抽象、转换后呈现出来。此举打破了实体结构间的不可切割的障碍，使用户可以用比原本形态更好的方式来应用这些资源。虚拟化资源一般包括计算能力和资料存储介质，这些资源的虚拟部分不受现有资源的架设方式、地域或物理形态所限制。虚拟化技术在带来成本节省和运维便利的同时，也带来了一些新的挑战，对架构师、运维人员、程序员等角色提出了新的要求。在解决了物理设施的虚拟化问题之后，我们的目光可能会转移到应用程序本身上来，因为这才是真正为用户体现支付价值的关键所在。特别需要注意的是，部署在虚拟化环境上的 Java 应用与物理环境上的应用存在明显的区别。

综上所述，性能优化本身对于程序性能是至关重要的，同时性能优化也是一门综合性课程，虽然本书针对的是 Java 程序的性能优化，但是依然需要考虑综合性因素。作者认为，随着 IT 技术的蓬勃、快速发展，性能调优已经不单纯是代码级别的调优，它是一个对综合性知识的深入理解需求，我们只有结合多方面的技术才能真正找到合理的解决方案。这也是本书除了深入介绍 Java 程序调优、JVM 调优等之外，坚持引入服务器、网络、云计算、虚拟化等多维技术点的原因。

1.2 性能优化的参考因素

系统性能优化，或者称之为程序性能优化，它存在的理由有很多。举一个政治上的例子，意大利由于天生的漫长海岸线，所以它一直以来都是难民逃亡欧洲的跳板。意大利政府一直都受困于这个难民潮问题，不管阻拦还是放行难民，这些举措都会受到北欧国家的指责。后来意大利政府从很多难民口中知道他们其实想去德国，只不过路过这里，所以干脆就来一招狠的，让难民填写意愿国家，只要填写了就直接大巴送到国境线上去。这样一来，德国吃紧了，一下子很多难民涌入，就造成了整个国家的运转问题。就好像计算机程序的性能问题，面对海量数据或者任务时，无论如何你都会碰到性能压力，唯一的选择是你会把这个压力放在哪一层或者哪一个位置来应对，以及采取什么应对措施。下面开始具体解释这些造成性能问题的因素点。

1.2.1 传统计算机体系的分歧

如果说图灵¹⁵奠定的是计算机的理论基础，那么冯·诺依曼¹⁶则是将图灵的理论物化为实际的物理实体，成为了计算机体系结构的奠基者。从第一台冯·诺依曼计算机诞生到今天已经过去了将近 70 年，计算机的技术与性能也都发生了巨大的变化，但整个主流体系结构依然是冯·诺依曼结构。

冯·诺依曼体系结构是采用二进制形式存储数据，硬件由 5 个部分组成，分别是运算器、控制器、存储器、输入设备和输出设备。同时提出了“存储程序”原理，即使用同一个存储器，然后经由同一个总线传输，程序和数据统一存储，同时在程序控制下自动工作。特别要指出，它的

¹⁵ 艾伦·麦席森·图灵 (Alan Mathison Turing)，生于 1912 年 6 月 23 日，逝于 1954 年 6 月 7 日，被誉为“计算机科学之父”和“人工智能之父”。图灵和同事破译的情报，在盟军诺曼底登陆等重大军事行动中发挥了重要作用，图灵因此在 1946 年获得“不列颠帝国勋章”。历史学家认为，他让二战提早了 2 年结束，拯救了至少 1400 万人的生命。

¹⁶ 冯·诺依曼 (John von Neumann, 1903—1957)，20 世纪最重要的数学家之一，在现代计算机、博弈论和核武器等诸多领域内有杰出建树的最伟大的科学全才之一，被称为“计算机之父”和“博弈论之父”。

程序指令存储器和数据存储器是合并在一起的，程序指令存储地址和数据存储地址指向同一个存储器的不同物理位置。因为程序指令和数据都用二进制码表示，且程序指令和被操作数据的地址又密切相关，所以几十年前选择这样的结构是合理的。

但是，随着对计算机处理速度要求的提高和对需要处理数据的种类、量级的需求不断增大，这种指令和数据共用一个总线的结构，使得信息流的传输成为限制计算机性能的一个瓶颈，制约了数据处理速度的提高。由此，体现出了冯·诺依曼体系结构的局限性，如下面 4 点：

- (1) 目前 CPU 的处理速度和内存容量的增长速率要远大于两者之间的流量，将大量数值从内存搬入搬出的操作占用了 CPU 大部分的执行时间，也造成了总线的瓶颈；
- (2) 程序指令的执行顺序是串行的，由程序计数器控制，这样使得即使有关数据已经准备好了，也必须遵循逐条执行指令序列，这样的设计影响了系统运行的速度；
- (3) 存储器是线性编址，按顺序排列的地址访问，这样设计有利于存储和执行机器语言，适用于数值计算。高级语言的存储采用的是一组有名字的变量，是按名字调用变量而不是按地址访问，且高级语言中的每个操作对于任何数据类型都是通用的，不管采用何种数据结构，多维数组¹⁷、二叉树¹⁸还是图，最终在存储器上都必须转换成一维的线性存储模型进行存储。这些因素都导致了机器语言和高级语言之间存在很大的语义差距，这些语义差距之间的映射大部分都要由编译程序来完成，在很大程度上增加了编译程序的工作量；
- (4) 冯·诺依曼体系结构计算机是为逻辑和数值运算而诞生的，它以 CPU 为中心，I/O 设备与存储器间的数据传送都要经过运算器，在数值处理方面已经达到很高的速度和精度，但对非数值数据的处理效率比较低，需要在体系结构方面有革命性突破。

科学家们一直在努力突破传统的冯·诺依曼体系结构框架，对冯·诺依曼计算机进行改良，主要体现在以下 3 点：

- (1) 将传统计算机只有一个处理器串行执行改成多个处理器并行执行，依靠时间上的重叠来提高处理效率，形成支持多指令流、多数据流的并行算法结构；
- (2) 改变传统计算机控制流驱动的工作方式，设计数据流驱动的工作方式，只要数据准备好了，就可以并行执行相关指令；
- (3) 跳出采用电信号二进制范畴，选取其他物质作为执行部件和信息载体，如光子、量子或生物分子等。

近几年，在计算机体系结构研究方面也已经有了重大进展，越来越多的非冯式计算机相继出现，如光子计算机、量子计算机、神经计算机以及 DNA 计算机等。

光子计算机 (Photonic Computer) 是一种采用光信号作为物质介质和信息载体，依靠激光束进入反射镜和透镜组成的阵列进行数值运算、逻辑操作和信息的存储和处理。它可以实现对复杂度高、计算量大、实时性强的任务的高效、并行处理，比普通电子计算机快 1000 倍，在图像处理、

¹⁷ 二维数组以上的数组，既非线性也非平面的数组。

¹⁸ 在计算机科学中，二叉树是每个节点最多有两个子树的树结构。通常子树被称作“左子树”(left subtree)和“右子树”(right subtree)。二叉树常被用于实现二叉查找树和二叉堆。

模式识别和人工智能方面有着非常巨大的应用前景。

神经计算机 (Neural Computer) 是一种可以并行处理多种数据功能的神经网络计算机, 它以神经元为处理信息的基本单元, 将模仿大脑神经记忆的信息存放在神经元上。神经网络具有自组织、自学习、自适应及自修复功能, 可以模仿人脑的判断能力和适应能力。美国科学家研究出的神经计算机可以模拟人的左脑和右脑, 能识别语言文字和图形图像, 能控制机器人行为, 进行智能决策。它的左脑由 100 万个神经元组成, 用于存储文字和语法规则, 右脑由 1 万多个神经元组成, 适用于图形图像识别。这有可能成为人工智能硬件发展的主攻方向。

量子计算机 (Quantum Computer) 是一种遵循量子力学规律进行高速数学和逻辑运算、存储及处理量子信息的物理装置。量子计算机本身的特性, 扩充了逻辑和数学理论, 通过核自旋、光子、束缚离子和原子等制成的量子位, 创造出了经典条件下不可能存在的新的逻辑门。与经典的比特位不同, 对量子位操作 1 次等同于对经典位操作 2 次, 因为量子不像半导体只能记录 0 和 1, 它可以同时表示多种状态。这些都为新的算法实现提供了条件, 也为人工智能的发展提供了可能的硬件条件。

我们不可否认, 冯·诺依曼计算机以其技术成熟、价格低廉、软件丰富和大众的使用习惯, 可能在今后很长的一段时期里还将为人类的工作和生活发挥着重要作用。但是, 为了满足人们对计算机更快速、更高效、更方便的使用要求, 为了让计算机能够模拟人脑神经元和脑电信号脉冲这样复杂的结构, 我们需要突破现有的体系结构框架并寻求新的物质介质作为计算机的信息载体, 这样才能使计算机有质的飞跃。相信未来随着非冯式计算机的商业化推进, 我们将会迎来一个崭新的信息时代。

1.2.2 导致系统瓶颈的计算资源

根据应用程序的不同特点, 任何计算机部件都有可能成为系统瓶颈爆发点。其中, 最有可能成为系统瓶颈的计算资源如图 1-1 所示, 包括 CPU、内存、磁盘、网络、数据库等。

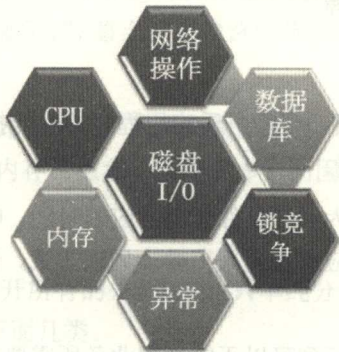


图1-1 系统瓶颈原因

- CPU: 对计算资源要求较高的应用, 由于其长时间、不间断地大量占用 CPU 资源, 这样对 CPU 的争夺将导致性能问题。如视频分析、科学计算、3D 渲染等应用场景都对 CPU 资源需求较大。
- 内存: 一般来说, 只要应用程序设计合理, 内存在读写速度上不太可能成为性能瓶颈, 除非应用程序进行了高频率的内存交换和扫描, 但这些情况比较少见。内存制约系统性能的最可能发生的情况是内存大小不足, 这种情况下会导致应用程序无法创建对象, 更严重甚

至导致操作系统无法正常运行。与磁盘相比，内存的大小较小，这意味着应用软件只能尽可能将常用的核心数据读入内存，大量的数据还是需要存放在磁盘上，这个特性在一定程度上降低了系统性能。

- **磁盘 I/O**：磁盘 I/O 读写速度比内存慢很多。随着硬盘技术的不断发展，SSD¹⁹固态硬盘的引入确实已经加快了磁盘的读写速度，但是性价比不高，速度也还是慢于内存。因为读写性能原因，程序在运行过程中，如果需要等待磁盘 I/O 完成，那么低效的 I/O 操作会拖累整个系统。
- **网络传送**：对网络数据进行读写的情况与磁盘 I/O 类似。由于网络环境的不确定性，尤其是对互联网上数据的读写，网络操作的速度可能比本地磁盘 I/O 更慢。因此，如果不加特殊处理，也极有可能成为系统瓶颈。
- **数据库**：大部分应用程序都离不开数据库，无论是关系型数据库，还是列式数据库，它们都存在连接数量、读写速度、数据合并等制约因素，而针对海量数据的读、写操作则可能更加耗费时间。应用程序可能需要等待数据库操作完成或者等待返回检索请求需要的结果集，即这类同步操作容易成为系统瓶颈。我们可以通过一些异步操作、多数据中心等方式来解决局部问题，但是无法解决所有由于数据库导致的问题。总的来说，数据库是最容易导致应用程序性能瓶颈的原因之一。
- **锁竞争**：对高并发程序来说，如果存在激烈的锁竞争，无疑是对性能极大的打击。锁竞争将会明显增加线程上下文切换的开销，而且这些开销都是与应用需求无关的系统开销，导致大量占用宝贵的 CPU 资源，却不带来任何好处。总的来说，锁竞争问题对于程序员来说最难处理，因为处理这方面问题需要大量的操作系统、编程语言并发知识及经验。
- **异常**：对 Java 应用来说，异常的捕获和处理是非常消耗资源的。如果程序高频率地进行异常处理，则整体性能便会有明显下降。高级编程语言一般都提供异常捕获及处理机制，相对来说程序员比较容易掌握。

1.2.3 程序性能衡量指标

如果抛开所有的内部技术因素，我们只看应用程序的性能指标，那么一般来说，程序的性能大体可以通过以下几个方面来衡量。

- **响应时间**：系统对用户行为或者事件做出响应的的时间。响应时间越短，性能一定越好，所以我们在系统设计过程中应该尽量采用异步处理方式，让用户能够尽快收到回执，这样用户体验会较好。
- **启动时间**：应用系统从运行到可以正常处理业务所需要花费的时间，对于用户来说，肯定是越快启动越好，所以我们在系统设计过程中应该尽量采用异步加载数据的方式启动应用程序，避免等待所有数据加载完毕后才启动。
- **执行时间**：一段代码从开始运行到运行结束，所使用的时间称为执行时间。对于执行时间，

¹⁹ Solid State Drives，简称固态硬盘，固态硬盘（Solid State Drive）用固态电子存储芯片阵列而制成的硬盘，由控制单元和存储单元（FLASH 芯片、DRAM 芯片）组成。

有些时候可能无法减少全局化的时间，但是可以通过把业务逻辑切分到多段连续的程序段中，让用户感觉执行时间减短了。

- **执行速度**：程序的反应是否迅速，响应时间是否足够短。该指标与响应时间、执行时间是相关联的。
- **计算资源分配**：计算资源，包括 CPU、内存、磁盘等，如果其中的任何一项分配不合理，可能会导致整个系统始终处于计算资源紧张的情况下，这样对于整个系统的性能影响一定是毁灭性的。
- **内存分配**：内存分配是否合理，是否过多地消耗内存或者存在泄漏，JVM 性能也与内存分配有一定关系。
- **磁盘吞吐量**：描述 I/O 的使用情况。IOPS (Input/Output Per Second) 即每秒的输入输出量（或读写次数），是衡量磁盘性能的主要指标之一。IOPS 是指单位时间内系统能处理的 I/O 请求数量，I/O 请求通常为读或写数据操作请求。随机读写频繁的应用，如 OLTP (Online Transaction Processing)，IOPS 是关键衡量指标。另一个重要指标是数据吞吐量 (Throughput)，指单位时间内可以成功传输的数据数量。对于大量顺序读写的应用，如 VOD (Video On Demand)，则更关注吞吐量指标。每秒 I/O 吞吐量 = IOPS × 平均 I/O SIZE。从公式可以看出，I/O SIZE 越大，IOPS 越高，那么每秒 I/O 的吞吐量就越高。因此，我们会认为 IOPS 和吞吐量的数值越高越好。实际上，对于一个磁盘来讲，这两个参数均有其最大值，而且这两个参数也存在着一一定的关系。
- **网络吞吐量**：描述网络的使用情况。网络中的数据由一个个数据包组成，防火墙对每个数据包的处理要耗费资源。吞吐量是指在没有帧丢失的情况下，设备能够接受的最大速率。其测试方法是：在测试中以一定速率发送一定数量的帧，并计算待测设备传输的帧，如果发送的帧与接收的帧数量相等，那么就将发送速率提高并重新测试；如果接收的帧少于发送的帧则降低发送速率重新测试，直至得出最终结果。吞吐量测试结果以“比特/秒”或“字节/秒”表示。
- **负载承受能力**：当系统压力上升时，系统的执行速度、响应时间的上升曲线是否平缓。负载承受能力与计算资源、内存、磁盘、网络等多方面因素都有关联。

1.2.4 性能优化目标

与前面章节不同，我们这里抛开所有的外部因素，只单纯分析 Java 程序本身，那么大多数 Java 程序性能优化目标都可以归纳到下面几类。

- **编写更有效率的代码**：应用程序的每 CPU 指令时钟周期 (Clocks Per Instruction, CPI) 指的是执行一条 CPU 指令所消耗的 CPU 时钟数。编译器将 Java 源代码编译成字节码，而 CPI 正是被用来衡量字节码效率的指标。由于应用程序的最终执行基于字节码，所以调整应用程序、Java 虚拟机或操作系统，缩短应用程序的 CPI，让编译器生成更优的指令等，这些举措都有助于提升应用程序的性能。执行路径长度与 CPI 之间有微妙的差别，执行路径长度与应用程序的算法选择关系密切，而 CPI 与编译器生成更有效的代码有关。前者着眼于通过选择合理的算法生成最短的 CPU 指令序列，后者着眼于让编译器生成最高效的代码，减少每条 CPU 指令上消耗的 CPU 时钟周期数。如果 CPU 时钟周期被用来执行操

作系统指令或内核代码，那么这部分时钟周期就无法用于执行应用程序。因此，改善应用程序性能的策略之一是减少消耗在系统或内核 CPU 上的时钟周期数。注意，该策略不适用于在操作系统或内核态上消耗时间极少的应用程序。

- **使用更高效的算法：**对应用程序进行性能调优所获得的最大收益来自于算法效率的提高。高效的算法主要可以在业务逻辑处理层面起到重要的作用，可以让应用程序使用更少的 CPU 指令、更短的执行路径来实现程序功能。通常情况下，拥有更短执行路径的应用程序运行得更快。从应用程序来看，使用更好的数据结构或者改进算法可以构造出更短的执行路径，很多应用程序的性能问题都源于使用了不合适的数据结构。因此，使用恰当的数据结构及算法是提升程序性能最有效的方法。在性能分析过程中，我们要充分注意程序使用的数据结构及算法，尽可能采用更优的方式，这样做才能最大限度地提高程序性能。
- **减少锁竞争：**对共享资源的竞争容易限制应用程序的可扩展性。频繁进行锁竞争的应用程序的性能是无法随线程数和 CPU 数增加而提高的，因此，减少锁竞争的频率、缩短锁持有的时间都能够有效地优化应用程序。

1.2.5 性能优化策略

上面各子章节对性能调优可能出现的位置、性能衡量指标、优化目标等进行了一些总结，下面列举几点常见的性能优化策略。

- **用空间换时间。**该策略属于系统架构层面的优化。我们知道，各种缓存机制，从 CPU L1/L2/RAM 到硬盘，都可以通过空间换时间的策略。这类策略基本上是通过采用把计算的过程一步一步地保存或者缓存等方式，避免重复计算的发生。具体的方式可以采用数据缓冲、CDN 等。类似的策略还表现为诸如冗余数据，比如数据镜像、负载均衡等。
- **用时间换空间。**该策略也属于系统架构层面的优化。有时候使用少量的空间，可能性能会更好。比如网络传输，如果有一些压缩数据的算法，例如 Huffman 编码压缩算法²⁰，该算法本身运行过程其实很耗时，但是因为整体来看性能瓶颈在网络传输部分，所以用时间来换空间反而能省时间。
- **简化代码。**该策略属于基础技术层面的优化。最高效的程序就是不执行任何代码的程序，所以，大多数情况下我们可以认为代码越少性能就越高。关于代码级优化的技术，大学的教科书中有很多示例了。比如，减少循环的层数、减少递归、在循环中少声明变量、少做分配和释放内存的操作、尽量把循环体内的表达式抽到循环外、条件表达式中的多个条件判断的次序、尽量在程序启动时把一些东西准备好、注意函数调用的开销（栈上开销）、注意面向对象语言中临时对象的开销、小心使用异常（不要用异常来检查一些可接受可忽略并经常发生的错误），等。这类知识需要我们非常了解编程语言和常用的语言自带资源库。从根本上来讲，不一定越少的代码越好，我们需要了解 Java 本地库的实现原理，例如同步问题，如果单纯采用同步锁（Synchronized）的方式，代码确实很少，但是实际的性

²⁰ 一种经典的压缩算法。于 1952 年问世，迄今为止仍经久不衰，广泛应用于各种数据压缩技术中，且仍不失为熵编码中的最佳编码方法，deflate 等压缩算法也结合了 huffman 算法。

能并不好，当大量线程同时访问代码块时，它会生成大量的锁，导致系统内部代码块互相等待，具体我们会在第 5 章详细介绍。所以，我们还是要有一定技术基础之后才能做到代码上的简化。

- **并行处理。**该策略属于一种综合性策略。试想，如果 CPU 只有一个核，你要是还采用多进程、多线程的方式编写代码，那么计算密集型需求的应用程序反而会更慢，这是由巨大的操作系统调度和切换开销所导致的，这类型优化策略只能依靠多核 CPU 才能真正体现出多进程多线程的优势。实际应用过程中，我们会针对不同 CPU 进行大量高并发、密集型任务测试，不一定核数多就一定是最佳选择，也有些情况需要单核能力强的 CPU，然后采用资源隔离技术对 CPU 核上的计算资源进行切分，把应用程序线程部署到不同的隔离区域，这样可以保证更多的并行程序同步进行，这也与整个系统的架构、内存共享策略、任务调度机制等多方面因素相关联。并行处理需要我们的程序拥有扩展性，不能水平或垂直扩展的程序无法进行并行处理。

综上所述，我们把经典的二八原则²¹移植到性能优化上来看，如图 1-2 所示，统计学认为，20%的系统设计或程序代码消耗了 80%的系统性能。如果我们可以找到那 20%的缺陷设计或者劣质代码，那么我们就可以较为容易地优化、解决 80%的性能问题。

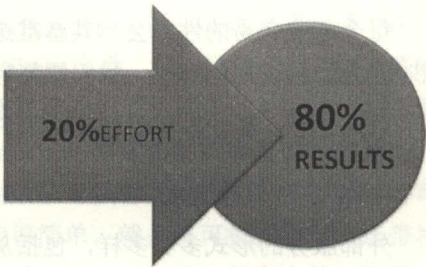
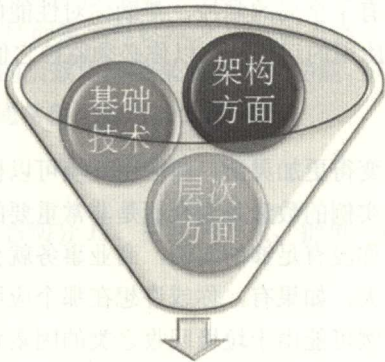


图1-2 二八原则

1.3 性能调优分类方法

性能调优方法一般来说可以分为基础技术、架构、层次这三个方面，如图 1-3 所示，解决了这三个方面的问题，沙漏才能正常工作。



性能调优分类方法

图1-3 性能调优分类方法

²¹ 即巴莱多定律，是 19 世纪末 20 世纪初意大利经济学家巴莱多发现的。他认为，在任何一组东西中，最重要的只占其中一小部分，约 20%，其余 80% 尽管是多数，却是次要的，因此又称二八定律。

1.3.1 业务方面

1.3.1.1 商业事务

商业事务是真实用户体验的直观反映，它们在抓取用户与应用交互时用户体验到的实时性能数据。测量商业事务的性能，需要抓取一件商业事务整体的响应时间及其各个组件的响应时间。这些响应时间再与满足业务需求的基准进行比较，从而决定应用是否正常。

商业事务通过其入口进行辨别，即它是用户与你的业务进行互动的入口。这类互动包括，一个网页请求、一个网页服务调用、消息队列中的一条消息等。当然，你也可以基于一个 URL 参数为同样的网页请求定义多个入口，或基于一个服务调用的内容定义多个入口点。关键在于，商业交易必须与你的业务流程相关联，比如说中国移动的空中缴费业务对应到系统中是多个原子服务，我们就应该将这几个原子服务通过相应的关联聚合成一个空中缴费业务来进行监控。

每个商业交易的性能会与其基准进行比较，判定其是否正常。譬如，如果某个商业事务的响应时间大于你设定的阈值，我们便判定其运行异常。

总而言之，商业事务最能反映用户体验，因此它们也是最重要的抓取维度。

1.3.1.2 外部服务

外部服务的形式多种多样，包括从属的网页服务、遗留系统或数据库等。外部服务是与应用交互的系统。运行在外部服务系统中的代码常常无法控制，但是我们可以控制这些系统的配置，因此了解他们是否运行正常以及何时出错也很重要。此外，我们必须有能力区分问题是出自应用本身，还是出自这些外部服务系统。

从商业事务的角度来说，我们可以辨别并测量这些处于自身应用的外部服务。例如，我们需要配置监控方法从而辨别那些包裹了外部服务调用的方法。但是对于常见的协议，诸如 HTTP 和 JDBC，外部服务可以自动检测。

商业事务让你对应用的性能有了全局的掌控，帮助你对性能问题进行分类。但是外部服务总能以意想不到的方式极大地影响应用的运行，所以你必须监控它们。

1.3.1.3 应用布局

因为云的出现，现在的应用变得更加灵活，即应用环境可以根据用户需求调节大小。因此，对应用的布局进行检测从而决定实例的数量是否合适是非常重要的。如果你的应用实例太多，你的云主机成本就会增加。但如果你没有足够的实例，商业事务就会受到影响。具体来说，你需要确定是否有应用中的实例负载过大，如果有，你或许想在那个应用中添加实例。从应用的角度查看实例状态很重要，因为单个实例可能由于垃圾回收之类的因素负载过大，但如果应用中大多数实例都负载过大，则该应用可能已经无法支持它接受的访问量。

1.3.2 基础技术方面

无论你现在正在专注于前端软件技术，还是后端软件技术，所有的语言都需要使用到编译器，那么一般来说编译器是如何工作的呢？

编译器在开始工作之前，需要知道当前的系统环境，比如标准库在哪里、软件的安装位置在哪里、需要安装哪些组件等。这是因为不同计算机的系统环境不一样，通过指定编译参数，编译

器就可以灵活适应环境，编译出各种环境都能运行的机器码。这个确定编译参数的步骤，就叫作“配置”（configure）。源码肯定会用到标准库函数（standard library）和头文件（header）。它们可以存放在系统的任意目录中，编译器实际上没办法自动检测它们的位置，只有通过配置文件才能知道。接下来，就是从配置文件中知道标准库和头文件的位置。一般来说，配置文件会给出一个清单，列出几个具体的目录。等到编译时，编译器就按顺序到这几个目录中，寻找目标。

对于大型项目来说，源码文件之间往往存在依赖关系，编译器需要确定编译的先后顺序。假定 A 文件依赖于 B 文件，编译器应该保证做到下面两点。

（1）只有在 B 文件编译完成后，才开始编译 A 文件。

（2）当 B 文件发生变化时，A 文件会被重新编译。

不同的源码文件，可能引用同一个头文件（比如 `stdio.h`²²）。编译的时候，头文件也必须一起编译。为了节省时间，编译器会在编译源码之前，先编译头文件。这保证了头文件只需编译一次，不必每次用到的时候都重新编译了，Java 也是类似的，称为 Class 文件。预编译完成后，编译器就开始替换掉源码中 `bash` 的头文件和宏。预处理之后，编译器就开始生成机器码。对于某些编译器来说，还存在一个中间步骤，会先把源码转为汇编码（assembly），然后再把汇编码转为机器码。机器码生成后连接是在内存中进行的，即编译器在内存中生成了可执行文件。下一步，必须将可执行文件保存到用户事先指定的安装目录。表面上，这一步很简单，就是将可执行文件（连带相关的数据文件）拷贝过去就行了。但是实际上，这一步还必须完成创建目录、保存文件、设置权限等步骤。这整个的保存过程就称为“安装”（Installation）过程。最后我们可以通过生成若干个二进制文件的形式，以开放服务把编译后的程序提供给用户。

1.3.2.1 前端技术

■ 前端负载均衡

通过 DNS²³的负载均衡器（一般在路由器上根据路由的负载重定向）可以把用户的访问均匀地分散在多个 Web 服务器上，这样的做法可以减少 Web 服务器的请求负载。因为 HTTP 的请求都是短连接，所以可以通过很简单的负载均衡器来完成这一功能。注意，最好是有 CDN 网络让用户连接与其最近的服务器（CDN²⁴通常伴随着分布式存储²⁵）。

■ 减少前端连接数

通过减少访问页面，CSS 静态切分 JS、图片的方式，把单一用户的连接数降到最低。

²² 以 C 语言为例。

²³ DNS（Domain Name System，域名系统），因特网上作为域名和 IP 地址相互映射的一个分布式数据库，能够使用户更方便的访问互联网，而不用去记住能够被机器直接读取的 IP 数串。

²⁴ CDN（Content Delivery Network，内容分发网络），其基本思路是尽可能避开互联网上有可能影响数据传输速度和稳定性的瓶颈和环节，使内容传输的更快、更稳定。

²⁵ 与目前常见的集中式存储技术不同，分布式存储技术并不是将数据存储在某一个或多个特定的节点上，而是通过网络使用企业中每台机器上的磁盘空间，并将这些分散的存储资源构成一个虚拟的存储设备，数据分散的存储在企业的各个角落。

■ 减少网页大小增加带宽

通过减少网页上附带的图片可以大大降低网络带宽压力。

■ 前端页面静态化

通过静态化一些不常变化的页面和数据，可以通过技术手段让用户直接从内存中读取这些信息，这样可以减少磁盘 I/O²⁶。

■ 优化查询

对于查询相同内容的用户，可以通过反向代理方式处理。这样的技术主要用查询结果缓存来实现，第一次查询时需要到数据库获得数据后把数据放到缓存，之后的查询直接访问高速缓存。

■ 缓存的问题

缓存可以被用来缓存动态页面，也可以被用来缓存查询的数据。很多 NoSQL²⁷数据库可以解决下述问题。

- (1) 缓存的更新，也叫缓存和数据库的同步。可以通过缓存失效时间限定、中心统一更新等方式实现；
- (2) 缓存的换页。通过把一些不活跃的数据换出内存可以解决内存不足的问题，常用的换页算法有 FIFO、LRU、LFU 等算法；
- (3) 缓存的重建和持久化。缓存一旦丢失，我们就需要重建缓存，如果数据量很大，缓存重建的过程会很慢，这会影响生产环境，所以，缓存的持久化也是需要被考虑的。

1.3.2.2 服务端技术

■ 数据冗余

通过减少表连接的方式可以降低数据冗余，但是它牺牲了数据的一致性，风险比较大。通过 NoSQL 可以冗余数据。

■ 数据镜像

几乎所有主流的数据库都支持镜像，数据库的镜像带来的好处之一是可以做负载均衡。把一台数据库的负载均分到多台上，同时又保证了数据一致性。最重要的是，这样还可以有高可用性，起到 HA²⁸的作用。

数据镜像的数据一致性可能是个复杂的问题，我们可以通过针对单条数据进行数据分区的方法

²⁶ 具体请参考有关 Linux 服务器的相应技术书籍。

²⁷ 泛指非关系型的数据库，NoSQL 数据库的产生就是为了解决大规模数据集合多重数据种类带来的挑战，尤其是大数据应用难题。

²⁸ 即 High Availability，指的是通过尽量缩短因日常维护操作（计划）和突发的系统崩溃（非计划）所导致的停机时间，以提高系统和应用的可用性。它与被认为是不间断操作的容错技术有所不同。HA 系统是目前企业防止核心计算机系统因故障停机的最有效手段。

式实现该功能。除了传统方式外，我们可以通过现在正火热的 Docker²⁹来实现数据镜像。

■ 数据分区

数据镜像不能解决的一个问题就是数据表里的记录太多，导致关系型数据库操作太慢，可以通过对数据进行分区来解决该问题。数据分区有很多种做法，一般来说有下面这几种：

- (1) 把数据把某种逻辑来分类。比如火车票的订票系统可以按各铁路局来分，可按各种车型分，可以按始发站分，可以按目的地分……反正就是把一张表拆成多张有一样的字段但是不同种类的表。这样，这些表就可以存在不同的机器上以达到分担负载的目的。
- (2) 把数据按字段切分。比如把一些不经常改的数据放在一个表里，经常改的数据放在另外多个表里。把一张表变成 1 对 1 的关系，这样，你可以减少表的字段个数，同样可以提升一定的性能。另外，字段多会造成一条记录的存储会被放到不同的页表里，这对于读写性能都有问题。但这样一来会有很多复杂的控制。
- (3) 平均分表。因为第一种方法是并不一定平均分均，可能某个种类的数据还是很多。所以，也有采用平均分配的方式，通过主键 ID 的范围来分表。
- (4) 同一数据分区。这个在上面数据镜像提过。也就是把同一商品的库存值分到不同的服务器上，比如有 10000 个库存，可以分到 10 台服务器上，一台上有 1000 个库存。

这四种分区都有好有坏。最常用的还是第 1 种。数据一旦分区，你就需要有一个或是多个调度来让你的前端程序知道去哪里找数据。

NoSQL 数据库因为它特有的分布式、数据互备概念，所以已经解决数据分区问题。

■ 后端系统负载均衡

数据分区可以在一定程度上减轻负载，但是还是无法减轻热点数据³⁰的负载，需要使用数据镜像来减轻负载。使用数据镜像，则必然要使用负载均衡，我们需要一个任务分配系统，该系统可以监控各个服务器的负载情况，这样的设计有点类似于 Master-Slaves，现在较为流行的大数据 Hadoop³¹、Spark³²等大数据框架都是按照这类方式设计的。

任务分配服务设计上天生有一些难点：

- 负载情况比较复杂。负载均衡算法设计没有固定规则，我们只有根据生产环境下整体的系统负载能力、单台机器的负载能力、外部等待任务等多方面的情况来设计该算法；
- 任务分配服务需要任务队列。任务队列可以帮助保持任务排队序列、收集任务执行状态、持久化任务；

²⁹ 一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。

³⁰ 数据的热点单点问题由于其独有的高访问特性，在性能上一直都是一大难题。

³¹ 一个由 Apache 基金会所开发的分布式系统基础架构。

³² UC Berkeley AMP lab 所开源的类 Hadoop MapReduce 的通用的并行，Spark，拥有 Hadoop MapReduce 所具有的优点；但不同于 MapReduce 的是 Job 中间输出结果可以保存在内存中，从而不再需要读写 HDFS，因此 Spark 能更好地适用于数据挖掘与机器学习等需要迭代的 MapReduce 的算法。

- 任务分配服务需要高可用性的技术。此外，我们还需要注意被持久化的任务队列如何转移到别的服务器上。

■ 异步

异步采用的是延时处理方式。在技术上来说，就是把各个处理程序做成并行化，这样也就可以水平扩展了。使用异步方式需要考虑几点：

- (1) 被调用方的结果返回，会涉及进程线程间通信的问题；
- (2) 如果程序需要回滚，回滚会有点复杂；
- (3) 异步通常都会伴随多线程多进程，并发的控制也相对麻烦一些；
- (4) 很多异步系统都用消息机制，消息的丢失和乱序也会是比较复杂的问题。

■ 节流阀

无论是叫节流阀或者流控机制，该技术主要是防止系统被超过自己最大负荷的外部调用拖垮，它属于一种对自身系统的保护机制。使用节流阀技术一般来说是针对一些自己无法控制的系统，比如，和 B2C 网站对接的银行系统，视频分析领域针对视频的流控措施均属于这类功能。

■ 批量处理

批量处理的技术是把一堆属于相同类别的信息放在一起请求批量处理的过程，它适用于离线处理模式，即适用于实时性要求不高的需求。网络上的 MTU（最大传输单元），以太网是 1500 字节，光纤可以达到 4000 多个字节，如果你的一个网络包没有放满这个 MTU，那就是在浪费网络带宽，因为网卡的驱动程序只有一块一块地读效率才会高。因此，网络发包时，我们需要收集到足够多的信息后再做网络 I/O，这也是一种批量处理的方式。

■ 垃圾回收技术

从 Java 发布最早版本开始，一直都保留的核心特性就是垃圾回收，垃圾回收使我们不再需要手动管理内存。当使用完一个对象后，我们只需删除它的引用，然后垃圾回收就会自动释放它。垃圾回收为程序员们减少了分配、释放内存空间等烦琐步骤。

尽管垃圾回收达成了无须手动管理内存的目标，也防止了传统的内存泄露，但是作为代价，垃圾回收过程有时相当笨拙。注意，根据不同的 JVM，垃圾回收策略也有所不同。垃圾回收最大的敌人就是传说中的主要（major）或（full）垃圾回收。除了 Azul JVM³³，所有的 JVM 都存在这个问题。

当垃圾回收运行时，它会运行一项可达性测试（reachability test），它会创建一个由对象组成的根集合（root set），该集合包含每个运行线程中的直接可见对象。接着，它会探寻根集合中的对象涉及的其他对象，然后探寻这些对象涉及的对象，直到所有对象都被涉及。在这个过程中，它会记录（mark）下现时活动对象的内存地址，然后把不被使用的所有地址都扫除（sweep）。说得更恰当些，它会把没有根集合对象引用的内存都释放。最终，它会压缩、整理这些内存，这样新

³³ Azul Systems 公司推出的 JVM(取名 Zing)，Zing 引入了一系列优秀的垃圾收集机制，使得应用系统能尽情使用需要的内存容量，而不会让 pause time 太长。唯一的不足貌似 Zing 是非开源，但作为开源项目可以免费使用。

的对象才能获得内存分配。

JVM 分布式模型用于决定是在一个 JVM 还是多个 JVM 上执行 Java 程序。你可以根据其有效性、响应能力和可维护性来进行选择。当在多台服务器上运行 JVM 时，你也可以选择将多个 JVM 运行于一台服务器或者每台服务器运行一个 JVM。例如，对于每台服务器，你可以运行一个使用 8GB 堆内存的 JVM，也可以运行 4 个使用 2GB 的 JVM。你应该根据处理器内核的个数、程序的特性等多种因素来决定这个数量。当优先考虑响应能力时，使用 2GB 的堆内存会优于 8GB 的，原因是这样能在更短的时间内完成 Full GC。当然，8GB 的堆内存可以降低 Full GC 的频率。如果你的程序使用了内部缓存，还可以通过增加缓存命中率来提高响应能力。综上所述，选择合适的模型需要考虑应用程序的特性，然后在各种模型中选定一个能够扬长避短的。

此外，选择 JVM 其实就是决定使用 32 位还是 64 位的 JVM。在相同的条件下，你最好用 32 位的。因为 32 位的 JVM 比 64 位性能更好。然而，32 位 JVM 最大支持的堆内存是 4GB（无论在 32 位操作系统还是 64 位的上，实际可分配的大小都只有 2~3GB）。如果需要更大的堆内存，还是用 64 位的 JVM 比较合适，未来的趋势也是 64 位。

1.3.3 组件方面

随着 IT 技术、产品的不断发展，目前主流的前沿技术包括云计算、虚拟化、CPU 多核、异构架构、高速网络架构等，如图 1-4 所示。

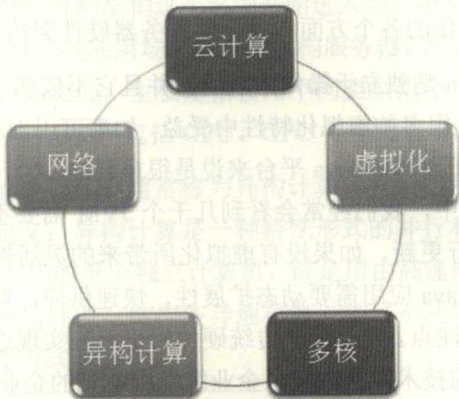


图1-4 架构方面涉及的技术

1.3.3.1 云计算

云计算是继 1980 年代大型计算机到客户端-服务器的大转变之后的又一种巨变。几年前，业界对于虚拟化和云计算是否能够在商业上真正应用还有很多争论，但目前，云计算和虚拟化已经被广泛应用于蓬勃发展的互联网领域以及传统的企业级解决方案之中，软件开发和部署正在经历着剧烈的变革。

云计算（Cloud Computing）是基于互联网的相关服务的增加、使用和交付模式，通常涉及通过互联网来提供动态易扩展且经常是虚拟化的资源。云是网络、互联网的一种比喻说法。云计算甚至可以让你体验每秒 10 万亿次的运算能力，拥有这么强大的计算能力可以模拟核爆炸、预测气候变化和市场发展趋势。用户通过电脑、笔记本、手机等方式接入数据中心，按自己的需求进行运算。对于云计算的定义，现阶段广为接受的是美国国家标准与技术研究院（NIST）定义，即云

计算是一种按使用量付费的模式,这种模式提供可用的、便捷的、按需的网络访问,进入可配置的计算资源共享池(资源包括网络、服务器、存储、应用软件、服务),这些资源能够被快速提供,只需投入很少的管理工作,或服务供应商进行很少的交互。

总的来说,云计算的流行是一种大势所趋,但是将所有数据都存放在共有云是否安全?人们这种不安的心理引发了私有云,或者公司内部数据中心的流行。此外,它还滋生了另一种新的潮流:共有云和私有云的混合体。

1.3.3.2 虚拟化

实际上,在各种类型的企业级产品化应用中,Java 应用是进行虚拟化的最佳选择,因为它很容易取得成功。通过分享我们的经验,希望能够帮助你避免在虚拟化大规模 Java 平台时所遇到的一些困难。

本书包含了针对物理化和虚拟化 Java 平台的实验过程和结果,希望能够帮助读者在进行虚拟化之前,纠正他们在物理化 Java 平台中所遇到的问题。

很多 Java 应用开发人员很了解开发过程,他们知道如何编写 Java 代码以及如何对 JVM 进行调优。但是,这些信息通常只保存在开发人员的脑子里面,并没有分享给其他人员。通常,运行 Java 平台所需要的技能在开发人员和管理人员之间是分割的,没有一个人能够完全理解这两个方面。但是,这种孤立式的理解正在发生着变化,因为有更多的人开始去理解如何编写和部署 Java 代码、调优 JVM 并认识虚拟化的各个方面和复杂的服务器硬件架构。

从根本上来讲,因为 Java 是独立于操作系统的,并且它不依赖于任何硬件,所以它是完美的虚拟化可选对象。Java 也能从很多的虚拟化特性中受益,如高可用性和不停机迁移。虚拟化为 Java 平台所增加的这些灵活性对于通用的 Java 平台来说是很重要的,对于大规模的 Java 平台来讲更是如此。在大规模的 Java 平台中,我们经常会看到几千个 JVM 需要不断地进行管理,如启动、停止以及在不不停机的情况下进行更新。如果没有虚拟化所带来的灵活性,这种类型的管理活动无法适应如此大的规模。企业级 Java 应用需要动态扩展性、快速供应,对于如今的开发和运维团队来讲,HA 正成为日益重要的关注点。完全基于传统硬件的平台来实现这些需求会非常复杂并且成本高昂。虚拟化是一种突破性的技术,它减轻了企业组织中常见的企业级 Java 应用需求所面临的压力。

客户希望虚拟化的方式能够减少服务器的数量。与此同时,客户也希望借助合并的机会来合理化某个特定负载所使用的中间件组件的数量。中间件组件通常会运行在 JVM 之中,并且规模是成百上千个的 JVM 实例,所以这就提供了很多机会来进行 JVM 实例合并。因此,中间件的虚拟化会带来两次合并的机会,首先是合并服务器实例,然后是合并 JVM 实例。这种趋势得到了业界广泛认同,毕竟所有的 IT 厂商都在考虑合并所带来的成本节省问题。

本书不会大规模地、深入地介绍虚拟化技术,但是出于全面性考虑,本书会有针对性地引入虚拟化概念与 Java 结合的性能调优策略。

1.3.3.3 多核技术

随着多核、线程、异构技术等技术的不断发展,我们对于 Java 程序的优化方式已经不单单是单一的 JVM 调优、Java 代码编写优化,它已经快速地进入到综合因素优化层面。

单线程的 Java 应用程序无法充分利用现代 CPU 架构上额外的硬件线程。那些单线程应用必须按照多线程的方式重构才能并行工作。此外,很多 Java 应用程序都有单线程阶段或操作,尤其是初始化或启动阶段。通过并行执行程序、同时利用多个线程,这些 Java 应用程序的初始化或启动性能将大幅提升。

多核处理器是单枚芯片(也称为“硅核”),能够直接插入单一的处理器插槽中,但操作系统会利用所有相关的资源,将它的每个执行内核作为分立的逻辑处理器。通过在两个执行内核之间划分任务,多核处理器可在特定的时钟周期内执行更多任务。多核技术能够使服务器并行处理任务,而在以前,这可能需要使用多个处理器,多核系统更易于扩充,并且能够在更纤巧的外形中融入更强大的处理性能,这种外形所用的功耗更低、计算功耗产生的热量更少。多核技术是处理器发展的必然近 20 年来,推动微处理器性能不断提高的因素主要有两个:半导体工艺技术的飞速进步和体系结构的不断发展。半导体工艺技术的每一次进步都为微处理器体系结构的研究提出了新的问题,开辟了新的领域;体系结构的进展又在半导体工艺技术发展的基础上进一步提高了微处理器的性能。这两个因素是相互影响,相互促进的。

1.3.3.4 异构计算

异构计算技术从 20 世纪 80 年代中期产生,由于它能经济有效地获取高性能计算能力、可扩展性好、计算资源利用率高、发展潜力巨大,目前已成为并行/分布计算领域中的研究热点之一。伴随着 GPU 技术的不断发展、成熟,异构架构这个名词开始进入到密集型计算需求的应用程序范畴中,特别是针对视频分析、科学计算这些应用场景,异构架构服务器已经进入稳定发展期。异构计算的英文名称是 Heterogeneous Computing,主要是指使用不同类型指令集和体系架构的计算单元组成系统的计算方式,常见的计算单元类别包括 CPU、GPU 等协处理器、DSP、ASIC、FPGA 等。

在异构计算系统上进行的并行计算通常称为异构计算。人们已从不同角度对异构计算进行定义,综合起来我们给出如下定义:异构计算是一种特殊形式的并行和分布式计算,它或是能用同时支持 SIMD 方式和 MIMD 方式的单个独立计算机,或是用由高速网络互连的一组独立计算机来完成计算任务。它能协调地使用性能、结构各异地机器以满足不同的计算需求,并使代码(或代码段)能以获取最大总体性能方式来执行。

1.3.3.5 高速网络技术

在网络世界里,无论出现怎样的新技术,基础部分都不会太大的变化,无非是在某些地方对某些功能分而化之,或是恰恰相反,将某些地方的某些功能整而合之,使它们周而复始地聚散离合而已。正因为基础技术早已成型,才需要我们更深入、更扎实地掌握它们。只要掌握好基础部分,那么无论上层运行的是什么技术和设备,我们都能够沉着应对,绝不会乱了手脚。

现代网络设备已经非常成熟,各类解决方法也一样趋于成熟,所以我们完全可以基于不同的应用程序需求来设计适合自己应用程序使用的网络环境。

1.3.4 架构方面

自然世界中,先天有缺陷的生物总是容易被细菌病毒入侵,而健壮的生物更能抵抗细菌病毒的攻击,计算机系统也是一样,若有先天的架构设计安全缺陷,那么在面临网络攻击的时候,就更容易被入侵或者破坏,甚至因为设计架构的原因,有些漏洞完全没有办法修复。

一般来说,兼容性越好的架构越能适应未来变化的需要,但是兼容性设计也会带来严重的安全问题,例如苹果的 USB-C 接口设计³⁴。苹果的 USB-C 接口设计产生了一个必须推翻自己才能修复的漏洞,所以这是一个糟糕的设计。

我们通常认知是,如果能以最少的成本实现系统,那最能体现设计者水平。有些设计场景就暴露了安全问题,例如因为降成本设计导致服务器无法防范 CC 攻击的场景。很多设计者会将每台服务器的负载率设定得非常高,高达 50%~70%,希望减少服务器的部署以降低成本,但是在正常业务场景下就有这么高的负载,被 CC³⁵攻击的时候会很容易被瘫痪,安全专家也没有办法在服务器上实施安全策略以防御攻击。

设计时保证一定的服务器冗余,能够降低攻击开始阶段系统就被攻击到瘫痪的概率,也为 DDoS³⁶安全专家的安全防御策略提供计算资源。如果有条件的话,最好是把业务部署在云上,这样被攻击的时候可以动态增加服务器数量,既能节省成本也能保障攻击的时候能够有效的防护。

数据和代码不分离意味着数据可以被当成代码执行,而数据是可以由用户(攻击者)自己定义的,也就是说用户(攻击者)可以自定义在系统上执行的代码,那么攻击者可以构造木马代码,作为数据输入给系统,系统执行这些木马代码后,系统就会被攻击者控制,这样的设计将给系统带来极大的风险。

设计模式要求我们遵循对扩展开放,对修改封闭的设计原则。对于修改封闭,就是说外部可以调用系统的接口使用系统的功能,但是看不到系统内部实现的代码,也不能对内部实现的代码进行修改。这常常给设计者一种错觉,认为外部使用者不知道系统内部是怎么实现的,不知道存在的安全缺陷,从而放心大胆地在内部留下许多安全隐患。

综合上面的描述,不管是程序设计优化,还是系统架构,我们都需要把安全作为首要指标去考虑,各种遗留下来的安全隐患积累到一定程度,或者被某个导火索事件引起安全事故的大爆发,导致企业的经营遭受重创!不管是架构师、研发人员,或是安全人员,都不应该让企业走到这一步绝境,而应该在产品设计之初就避免出现严重的设计安全问题,为保障产品的安全水平打下基础。虽然没有绝对安全的系统,但是我们要向着这个目标不断去努力、发展。

1.3.5 层次方面

性能分析方法最常用的有两种方式,一种是所谓的自上而下,对应的,另一种是自下而上。自上而下指的是从应用层开始着手分析、查找性能瓶颈原因,相反地,自下而上指的是从分析 CPU、内存、磁盘 I/O 等底层问题开始,逐渐向上直到应用层,逐渐地分析、查找问题原因。由于硬件、操作系统、网络设备的不断发展,加之互联网应用群体的数量庞大,导致性能问题已经不太可能由单一原因导致,性能优化技术逐渐变化成综合性问题,对应的 Java 优化方案也需要综合性考虑整体原因后才能提出解决方案。无论自上而下,还是自下而上,不同的方法可以用来查找不同类

³⁴ 只要将特制的 U 盘插入使用 USB-C 接口的苹果计算机,攻击者无须在做任何操作,就可以将后门或是病毒自动植入苹果计算机,使计算机可以被攻击者远程控制或是通过病毒自动破坏计算机内的文件。

³⁵ CC (Challenge Collapsar) 是一种 http 层的 DDoS 攻击,它通过发送大量 http 层的请求,以达到让被攻击的目标网站瘫痪的目的。

³⁶ 分布式拒绝服务 (DDoS:Distributed Denial of Service) 攻击指借助于客户/服务器技术,将多个计算机联合起来作为攻击平台,对一个或多个目标发动 DDoS 攻击,从而成倍地提高拒绝服务攻击的威力。

型的性能问题，如图 1-5 所示。



图1-5 两种性能调优方法

1.3.5.1 自上而下

自上而下方法通常从应用层开始分析、查找问题的原因。由于每天使用应用层的人群数量不同、网络状态不同、应用层的配置可能不同等外部干扰因素，所以我们有必要对应用层软件进行监控，监控所有用户的使用过程，这样可以帮助发现是什么原因导致了性能下降。应用层的监控包括了 Java 虚拟机、中间件、数据库等，同时也需要监控操作系统的变化来帮助定位问题。基于这些监控数据，我们可以有效地对 Java 代码、JVM 配置选项、JVM 垃圾收集器等与 Java 相关的方面进行调优。

1.3.5.2 自下而上

自下而上方法是从最底层的硬件配置开始逐渐向上的方式进行分析、查找性能问题的原因。由于虚拟化技术的不同引入、强化，我们已经不再是对单一的 CPU、内存、硬盘、网络等资源进行管理、应用，我们可以通过虚拟化技术虚拟出一块资源，所以不同的虚拟资源可以带来不同的性能，这就需要对底层资源进行监控。监控的数据包括 CPU 指令集、CPU 缓存、磁盘缓存等。

1.4 本章小结

本章首先介绍了为什么我们需要性能调优，通过 12306、奥运会票务系统、B2C 网站等例子让读者能够明白性能优化的重要性和必要性。然后通过对性能的参考指标介绍，让读者了解性能优劣的评判依据。接下来，对性能调优进行分类，按照基础技术、系统架构、层次序列等三个方法进行性能优化。通过对本章的阅读，读者可能可以明确阅读本书的目的，然后通过第 2 章的预备知识，对性能调优开始前的基础技术有一定的了解。

第 2 章 优化前的准备知识

19 世纪 70 年代, 恩格斯根据当时自然科学发展所显示的突破原有学科界限的新趋势, 在分析各种物质运动形态相互转化的基础上指出, 原有学科的邻接领域将是新学科的生长点。此后在物理学、化学、生物学、地质学、天文学等原有基础学科相互交界的领域产生出了一系列的边缘学科, 如物理化学、生物化学、生物物理。笔者认为, 计算机软件设计也是一个可以归为“边缘科学”的学科, 它需要计算机技术与其他领域有联系后才能产生真正的价值, 这样这款软件才会拥有属于它自己的用户, 才会存在软件生命周期。

随着互联网时代的到来, 计算机需要管理的数据量呈指数级别地飞速上涨, 我们的系统所需要支持的用户数、数据量, 很可能在短短的一个月内突然爆发式地增长几千、几万倍, 数据也很可能快速地从原来的几百 GB 飞速上涨到了几百个 TB。如果在这爆发的关键时刻, 系统不稳定或无法访问, 那么对于业务将会是毁灭性的打击。这样看来, 计算机软件技术已经不再是单一的技术, 它已经发展成为整合了硬件、网络、系统架构、虚拟化、分布式计算诸多因素混合的一个综合性知识领域。因此, 我们对系统的性能优化也需要考虑诸多因素, 这也就需要我们对应用软件相关的多个领域的知识进行深入了解、理解。

基于 Java 平台的应用服务器、企业服务总线、消息中间件、流程引擎这些企业应用的关键运行平台还会在一段时间内被广泛使用。但是随着硬件技术的飞速发展, 以及新的应用模式和商业模式例如 SOA、云计算的出现和成熟, 面向企业应用的开发语言越来越需要关注并行计算、多核编程、极限事务处理等。例如金融行业, Java 正在逐步走入金融核心领域, 很多集成商和行业最终用户都在基于 Java 和 SOA 做银行的新一代核心。而且轻量级的 IOC 容器、OSGi 的应用服务期已经逐步成为主流, 尤其是在云计算的大环境下, 企业应用逐渐互联网化, 因此云化是大势所趋。

计算机的发展创造了 Internet, 但是计算机现在却不是访问 Internet 的唯一方式。智能化的消费类电子产品打破了 PC 作为信息终端的垄断地位, 成为人类进入 Internet 的新门户。信息终端的多元化预示着所谓后 PC 时代的到来。消费类的信息终端量大面广, 是典型的瘦客户机, 其本身的资源和能力不能与 PC 相比, 但必须更加智能化, 并对服务器端的管理提出了更高的要求。基于

Java 的安卓操作系统能运行在 16 位以上的微处理器上, 占用内存少, 可以在资源有限的设备商方便地开发出各种各样的应用, 直接运行在不同的消费类或其他电子设备上。Jini 的出现为 Java 网络连接提供了公共标准, 使得任何 Java 设备都可以连入网络中被自动识别, 并可充分利用网络上已有的各种资源。计算的网络化、嵌入化、部件化这三大趋势是紧密联系的。Java 和作为 Java 扩展的 Jini 技术将为这三者的结合找到合适的纽带。Java 绝不仅仅是一种语言, Java 和作为 Java 扩展的 Jini 是一个分布式的, 部件化的, 可广泛运用于从服务器、PC 机到机顶盒、微波炉、智能卡等多种设备的、与操作系统无关的优秀网络计算平台。

Java 系统主要的展示有: 应用工具、应用系统、信息家电等。特别在实时系统开发方面, 以 IBM 为首开发出了应用于工业实时环境的 Java 嵌入系统, 展现出 Java 在工业领域的广阔应用前景。

Java 语言的出现和发展, 得到了 IT 业界的青睐。作为一种与底层硬件无关的, “编写一次、到处运行” 的高级语言和计算平台, Java 天生就具有将网络上的各个平台连成一体的能力, 优秀的多线程设计也是 Java 语言的一大特色, 多平台的支持是其他编程语言所无法比拟的。Java 语言最初并不是为网络环境设计的, 用户能用它编写独立的桌面应用程序, 在这个领域 Java 已经被广大厂商接受, 如 Oracle 数据库、Eclipse 工具都是使用 Java 语言编写的。当网络出现以后, 由于网络软硬件环境的复杂性, 常见的编程语言不能适应这种环境的要求, 而 Java 语言的平台无关性的特性正好适用于网络这个潮流。

本章主要介绍和解决以下问题, 这些也是优化之前的准备知识:

- 什么是内存、CPU、GPU、硬盘、网络, Java 程序怎么样才能更好地利用它们。
- 那些高大上的技术, 集群技术、云计算技术、分布式技术、虚拟化技术, 它们是什么。
- 为第 3 章开始的具体编程、原理讲解做准备。

2.1 服务器知识

如果把社区比作一个云计算数据中心, 每个社区里面都建了很多房子, 这些房子有些是固定的, 有些是临时的, 临时的房子随时等待拆除, 房子和房子之间有很多的道路联通, 某些房子还被用于其他用途, 比如垃圾处理、发电, 诸如此类。我们可以把房子想象成为内存或者硬盘, 这取决于房子是否是固定的 (临时的是内存, 固定的是硬盘), 可以把道路想象成为网络, 把某些特定房子想象成为 CPU、GPU, 这些组成了我们本节需要讨论的知识, 即服务器相关知识。

2.1.1 内存

毫无疑问, 内存无论在最初的大型机, 还是在现代微型计算机世界, 都一直占据着至关重要的地位。任何处于运行过程中的程序或者数据都需要依靠内存作为存储介质, 否则程序将无法正常运行, 数据在持久化之前也没有暂时居住地点。与 C/C++ 语言相比, 使用 Java 语言编写的程序并不需要程序员显式地为每一个对象都编写对应的内存分配和内存回收等相关函数, 拥有这样的便捷主要是得益于 JVM 的自动内存管理机制, 让 Java 程序员可以从内存管理中解放出来, 只需要关注于自身业务即可, 这也就让一些程序员误认为 Java 语言门槛低, 稍加培训就可以上岗的错误观点。

尽管 JVM 的自动内存管理机制大大提升了 Java 程序员的编程效率, 甚至从某种意义上来说

降低了内存泄漏和内存溢出的风险，但是如果 Java 程序员过度依赖于自动化内存管理机制，那么最严重的就是会弱化 Java 程序员在程序出现内存溢出时定位及解决问题的能力，最终导致 Java 程序员完全无法自己处理内存泄漏这些严重性能缺陷。只有当我们真正了解 JVM 如何管理内存后，才能够在遇见 `OutOfMemory`¹ 错误时，快速地根据错误异常日志信息定位和解决问题。

除了下面要深入讨论的 JVM 内存区域以外，我们也可以直接访问内存，例如 JDK 在 NIO 类中实现了一种基于通道与缓冲区的 I/O 方式，它可以使用 Native 函数库²直接分配堆外内存，然后通过一个存储在 Java 堆中的 `DirectByteBuffer` 对象（既想要跟 JVM 相同进程内的存取，又希望不占用堆内存，可以选择采用 `DirectByteBuffer` 类）作为这块内存的引用进行操作。直接内存的分配不会受到 Java 堆大小的限制，但是会受到本机内存大小的限制，所有也可能会抛 `OutOfMemoryError` 异常。

我们来讨论一下 JVM 对内存区域的使用。JVM 的设计者们之所以将 JVM 的内存结构划分为多个不同的内存区，是因为每一个独立的内存区都拥有各自的用途，都会负责存储各自的数据类型，其中一些内存区的生命周期往往还会和 JVM 的生命周期一定程度上保持一致。也就是说，会伴随着 JVM 的启动而创建，伴随着 JVM 的退出而销毁。而另一部分内存区则是与线程的生命周期保持一致，会伴随着线程的开始而创建，伴随着线程的消亡而销毁。尽管不同的内存区在存储类型和生命周期上有一定的区别，却都拥有一个相同的本质，即存储程序的运行时数据。

Java 程序对内存分配的方式一般有三种：

- （1）从静态存储区域分配。内存存在程序编译的时候就已经分配好，这块内存存在程序的整个运行期间都存在。例如全局变量，`static` 变量。
- （2）在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。
- （3）在堆上分配，亦称动态内存分配。程序在运行的时候用 `malloc` 或 `new` 申请任意多少的内存，程序员自己负责在何时用 `free` 或 `delete` 释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。

Java 虚拟机内存模型是 Java 程序运行的基础。Java 虚拟机在执行 Java 的过程中会把它所管理的内存划分为若干不同的数据区域，这些区域都有各自的用途以及创建和销毁的时间。Java 虚拟机所管理的内存包几个运行时数据区域。为了能使 Java 应用程序正常运行，JVM 虚拟机将其内存数据分为程序计数器、虚拟机栈、本地方法栈、Java 堆和方法区这五个部分。如果根据受访权限的不同，我们可以定义上述几个区域分为线程共享和线程私有两大类。线程共享指的是可以允许被所有的线程共享访问的一类内存区，这类区域包括堆内存区、方法区、运行时常量池三个内存区。简单地一句话描述用途，如图 2-1 所示，程序计数器用于存放下一条运行的指令，虚拟机栈和本地方法栈用于存放函数调用堆栈信息，Java 堆用于存放 Java 程序运行时所需的对象等数据，方法区用于存放程序的类元数据信息。其中，Java 堆（heap）和 Java 栈（stack）这两个内存

¹ Out Of Memory（内存溢出）是一个程序员常见的错误类型。通常是开启应用程序过多所导致。

² Java 无法直接访问到操作系统底层（如系统硬件等），为此 Java 使用 native 方法来扩展 Java 程序的功能。

区是大多数 Java 程序员最关注的，两个英文单词连起来读，听着像 Hip-Hop³ 的感觉。

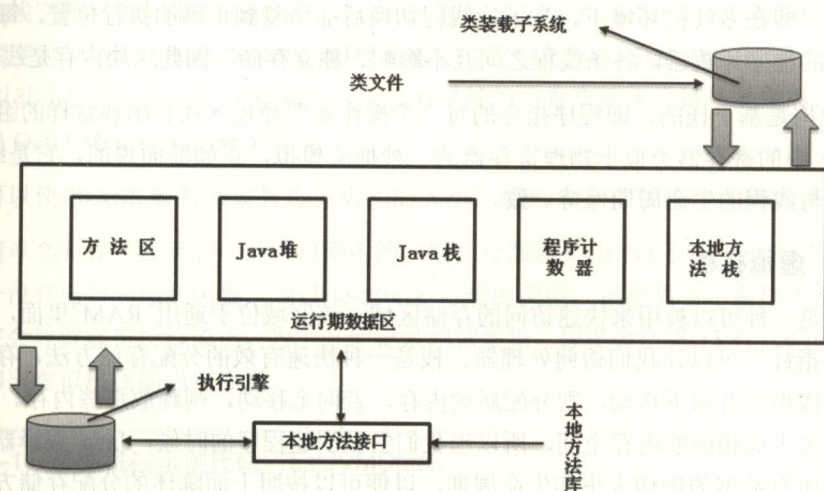


图2-1 JVM内存管理图

2.1.1.1 程序计数器

冯·诺伊曼计算机体系结构的主要内容之一就是“程序预存储，计算机自动执行”，处理器要执行的程序（指令序列）都是以二进制代码序列方式预存储在计算机的存储器中，处理器将这些代码逐条地取到处理器中再译码、执行，以完成整个程序的执行。为了保证程序能够连续地执行下去，CPU 必须具有某些手段来确定下一条取址指令的地址，程序计数器正是起到这种作用，所以通常又称之为“指令计数器”。

程序计数器（Program Counter Register）是一块很小的内存空间，它是运行速度最快的存储区域，因为它位于不同于其他存储区的地方——处理器内部。寄存器的数量极其有限，所以寄存器由编译器根据需求进行分配。实际在 JAVA 应用程序内部你不能直接控制寄存器，也不能在程序中感觉到寄存器存在的任何迹象。我们可以把程序计数器看作是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里，字节码解释器的工作就是通过改变程序计数器的值来选择下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都要依赖这个计数器来完成。

由于 Java 是支持多线程的语言，所以当线程数量超过 CPU 数量时，线程之间会自动根据时间片⁴轮询方式抢夺 CPU 资源。对于单核 CPU 而言，每一时刻只能有一个线程处于运行状态，其他线程必须被切换出去，直到轮询到自己才能使用 CPU 资源，为此，每一个线程都必须用一个独立的程序计数器，它被用来记录下一条需要执行的计算机指令。对于多核 CPU 来说，可以允许多个线程同时执行，各个线程之间的计数器互不影响、独立工作，所以程序计数器是线程独有的一块内存空间。如果当前线程正在执行一个 Java 方法，则程序计数器记录正在执行的 Java 字节码地址，如果当前线程正在执行一个 Native 方法，则程序计数器为空。根据 Java 虚拟机定义来看，程序寄

³ Hip-Hop 是起源于源自于街头的一种黑人文化，也泛指 rap（说唱乐）。

⁴ 即 CPU 分配给各个程序的时间，每个线程被分配一个时间段，称作它的时间片，即该进程允许运行的时间，使各个程序从表面上看是同时进行的。

存器区域是唯一一个在 Java 虚拟机规范中没有规定任何 `OutOfMemoryError` 情况的区域。简单概括上面的描述,即多线程环境下,为了让线程切换后能恢复到正确的执行位置,每条线程都需要有一个独立的程序计数器,各条线程之间互不影响、独立存储,因此这块内存是线程私有的。

JVM 的架构是基于栈的,即程序指令的每一个操作都要经过入栈和出栈这样的组合型操作才能完成。JVM 中的寄存器类似于物理寄存器的一种抽象模拟,正如前面说的,它是线程私有的,所以生命周期与线程的生命周期保持一致。

2.1.1.2 虚拟机栈

虚拟机栈是一种可以被用来快速访问的存储区域,该区域位于通用 RAM⁵里面,通过使用它的所谓的“栈指针”可以让我们访问处理器。栈是一种快速有效的分配存储方法,存取速度仅次于寄存器,堆栈指针若向下移动,则分配新的内存,若向上移动,则释放那些内存。由于 Java 编译器需要预先去生成相应的内存空间,所以当我们尝试创建程序的时候,Java 编译器必须知道被存储在栈内的所有数据的确切大小和生命周期,以便可以按照上面陈述的分配存储方法通过上下移动堆栈指针来动态调整内存空间。我们可以认为这一约束限制了程序的灵活性,所以这就是为什么只有某些 Java 数据,特别是对象引用,它被存储在栈里面,而应用程序内部数量庞大的 Java 对象没有被存储在虚拟机栈里面。

总的来说,栈的优势是存取速度比堆要快,它仅次于寄存器,并且栈数据是可以被共享的。栈的缺点是存储在栈里面的数据大小与生存期必须是确定的,从这一点来看,栈明显缺乏灵活性。虚拟机栈中主要被用来存放一些基本类型的变量,例如 `int`、`short`、`long`、`byte`、`float`、`double`、`boolean`、`char`,以及对象引用。

我们前面说过,虚拟机栈有一个很重要的特殊性,就是存在栈中的数据可以共享。假设我们同时定义:

```
int a = 1;
int b = 1;
```

对于上面的代码,编译器处理第一条语句,首先它会在栈中创建一个变量为 `a` 的引用,然后查找栈中是否有 1 这个值,如果没找到,就将 1 存放进来,然后将 `a` 指向 1。接下来处理第二条语句,在创建完 `b` 的引用变量后,因为在栈中已经有 1 这个值,便将 `b` 直接指向 1。这样,就出现了 `a` 与 `b` 同时均指向 1 的情况。这时,如果存在第三条语句,它针对 `a` 再次定义为 `a=4`;那么编译器会重新搜索栈中是否有 4 值,如果没有,则将 4 存放进来,并令 `a` 指向 4;如果已经有了,则直接将 `a` 指向这个地址。因此 `a` 值的改变不会影响到 `b` 的值。要注意这种数据的共享与两个对象的引用同时指向一个对象的这种共享的方式存在明显的不同,因为这种情况 `a` 的修改并不会影响到 `b`,它是由编译器完成的,这样的做法有利于节省空间。而一个对象引用变量修改了这个对象的内部状态,会影响到另一个对象引用变量。

与程序计数器一样,Java 虚拟机栈也是线程私有的内存空间,它和 Java 线程在同一时间创建,它保存方法的局部变量、部分结果,并参与方法的调用和返回。

⁵ 随机存取存储器 (random access memory, RAM) 又称作“随机存储器”,是与 CPU 直接交换数据的内部存储器,也叫主存(内存)。它可以随时读写,而且速度很快,通常作为操作系统或其他正在运行中的程序的临时数据存储媒介。

Java 虚拟机规范允许 Java 栈的大小是动态的或者是固定不变的。在 Java 虚拟机规范中定义了两种异常与栈空间有关，即 `StackOverflowError` 和 `OutOfMemoryError`。如果线程在计算过程中，请求的栈深度大于最大可用的栈深度，则程序运行过程中会抛出 `StackOverflowError` 异常。如果 Java 栈可以动态扩展，而在扩展栈的过程中没有足够的内存空间来支持栈的发展，则程序运行过程中会抛出 `OutOfMemoryError` 异常。

我们可以使用 `-Xss` 参数来设置虚拟机栈的大小，栈的大小直接决定了函数调用的可达深度。

代码清单 2-1 所示展示了一个递归调用的应用。计数器 `COUNT` 记录了递归的层次，`recursion` 方法是一个没有出口的递归函数，通过 `testStack` 方法的调用，该函数会不断地申请栈的深度，最终程序一定会导致虚拟机栈溢出。为了方便记录测试数据，程序会在第 13 行代码中当栈溢出异常发生时打印出虚拟机栈的当前深度。

代码清单 2-1 虚拟机栈示例代码 `TestJVMStack`

```
public class TestJVMStack {
    private int count = 0;
    //没有出口的递归函数
    public void recursion(){
        count++; //每次调用深度加 1
        recursion(); //递归
    }

    public void testStack(){
        try{
            recursion();
        }catch(Throwable e){
            System.out.println("deep of stack is "+count); //打印栈溢出的深度
            e.printStackTrace();
        }
    }

    public static void main(String[] args){
        TestStack ts = new TestStack();
        ts.testStack();
    }
}
```

清单 2-1 程序输出如清单 2-2 所示，笔者本机运行的程序最终在申请到 9013 层虚拟机栈时，抛出异常。

代码清单 2-2 代码 2-1 程序运行输出

```
java.lang.StackOverflowError
at TestStack.recursion(TestStack.java:7)
at TestStack.recursion(TestStack.java:7)
at TestStack.recursion(TestStack.java:7)
at TestStack.recursion(TestStack.java:7)
at TestStack.recursion(TestStack.java:7)
```



```
at TestStack.recursion(TestStack.java:7)
at TestStack.recursion(TestStack.java:7)deep of stack is 9013
```

如果系统需要支持更深的栈调用，则可以使用参数-Xss1M 运行程序，可以扩大虚拟机栈的大小，刚才的配置扩大栈空间的最大值为 1MB。

如图 2-2 所示，虚拟机栈在运行时使用一种叫作栈帧的数据结构保存上下文数据。栈帧里面存放了方法的局部变量表、操作数栈、动态连接方法和返回地址等信息。每一个方法的调用都伴随着栈帧的入栈操作，相应地，方法的返回则表示栈帧的出栈操作。如果方法调用时，方法的参数和局部变量相对较多，那么栈帧中的局部变量表就会比较大，栈帧会不断膨胀以满足方法调用所需传递的信息增大需求。因此，单个方法调用所需的栈空间大小也会比较多。

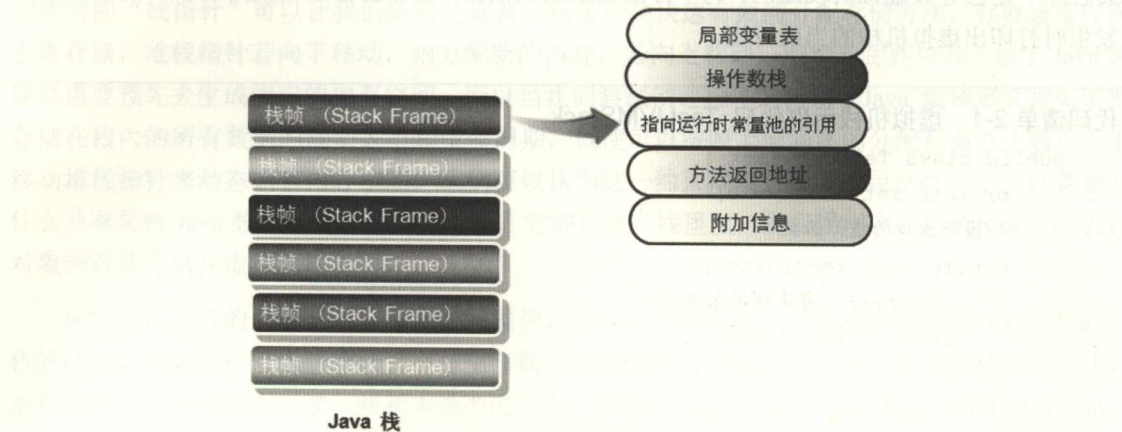


图2-2 虚拟机栈引用图

栈帧分为三部分，局部变量区 (Local Variables)、操作数栈 (Operand Stack) 和帧数据区 (Frame Data)。

局部变量区被定义为一个从 0 开始的数字数组，byte、short、char 在存储前被转换为 int，boolean 也被转换为 int，0 表示 false，非 0 表示 true，long 和 double 则占据两个字长。局部变量区是通过数组下标访问的。

操作数栈也被组织为一个数字数组，但不同于局部变量区，它不是通过数组下标访问的，而是能过栈的 Push 和 Pop 操作，前一个操作 Push 进的数据可以被下一个操作 Pop 出来使用。

帧数据区这部分的作用主要有以下三点。

- 解析常量池里面的数据。
- 方法执行完后处理方法返回，恢复调用方现场。
- 方法执行过程中抛出异常时的异常处理，存储在一个异常表，当出现异常时虚拟机查找相应的异常表看是否有对应的 Catch 语句，如果没有就抛出异常终止这个方法调用。

函数嵌套调用的次数由栈的大小决定。一般来说，栈越大，函数嵌套调用次数越多。对一个函数而言，它的参数越多，内部局部变量越多，它的栈帧就越大，其嵌套调用次数就会减少。

代码清单 2-3 所示代码相较于代码清单 2-1 所示代码，增加了 recursion 方法的参数，用以展示内部局部变量增多之后的变化。

代码清单 2-3 虚拟机栈示例代码 TestJVMStack1

```

public class TestJVMStack1 {
    private int count = 0;
    //没有出口的递归函数
    public void recursion(long a,long b,long c) throws InterruptedException{
        long d=0,e=0,f=0;
        count++;//每次调用深度加1
        recursion(a,b,c);//递归
    }

    public void testStack(){
        try{
            recursion(1L,2L,3L);
        }catch(Throwable e){
            System.out.println("deep of stack is "+count);//打印栈溢出的深度
            e.printStackTrace();
        }
    }

    public static void main(String[] args){
        TestStack ts = new TestStack();
        ts.testStack();
    }
}

```

代码清单 2-3 代码运行后的输出如清单 2-4 所示。

代码清单 2-4 代码 2-3 运行输出

```

deep of stack is 3432
java.lang.StackOverflowError
at TestStack.recursion(TestStack.java:8)

```

从代码清单 2-4 的输出，我们可以看到，随着代码传入了多个参数和局部变量，栈帧大小就会膨胀。

在栈帧中，与性能调优关系最为密切的部分就是前面提到的局部变量区。局部变量区被用于存放方法的参数和方法内部的引用。局部变量区以“字”为单位进行内存的划分，一个字为 32 位长度。对于 long 和 double 型的变量，则占用 2 个字，其余类型使用 1 个字。在方法执行时，虚拟机使用局部变量区完成方法的传递，对于非静态（static）方法，虚拟机还会将当前对象（this）作为参数通过局部变量区传递给当前方法。

使用 jclasslib⁶工具可以查看 class 文件中每个方法所分配的最大局部变量区的容量。jclasslib 工具是开源软件，它可以用于查看 class 文件的结构，包括常量池、接口、属性、方法，还可以用于查看文件的字节码。

⁶ jclasslib 不但是一个字节码阅读器而且还包含一个类库允许开发者读取、修改、写入 Java Class 文件与字节码。

局部变量区的基本单位“字”对系统 GC⁷也有一定影响。如果一个局部变量被保存在局部变量区里面，那么 GC 能引用到这个局部变量所指向的内存空间，从而在 GC 时无法回收这部分空间，如清单 2-5 程序所示。

代码清单 2-5 虚拟机栈示例代码 testGC

```
public class testGC {
    public static void test1()
    {
        byte[] a = new byte[6*1024*1024];
    }
    System.gc();
    System.out.println("first explicit gc over");

    public static void main(String[] args){
        testGC.test1();
    }
}
```

清单 2-5 代码中，第 4 行定义了一个局部变量 `a`，并且它的作用范围仅限于大括号中。在显示的 GC 调用时，变量 `b` 已经超过了它的作用范围，其对应的堆空间理应被回收。而事实上，由于变量 `a` 仍在该栈帧的局部变量区内，因此 GC 可以引用到该内存块，阻碍了其回收过程。

假设在该变量失效后，在这个函数体内又未能有定义足够多的局部变量来复用该变量所占的基本单位“字”，那么在整个函数体内部，这块内存区域是不会被 GC 回收的。如果函数体内的后续操作非常费时或者又申请了较大的内存空间，则对系统性能将会造成较大的压力。在这种环境下，可以通过手动给要释放的变量赋值为 `null` 的方法来解决这个潜在的性能问题。

2.1.1.3 本地方法栈

本地方法栈（Native Method Stacks）和 Java 虚拟机栈的功能很相似，Java 虚拟机栈用于管理 Java 函数的调用，而本地方法栈用于管理本地方法的调用。本地方法并不是用 Java 实现的，而是使用 C 实现的。当某个线程调用一个本地方法时，他就进入了一个全新的并且不再受虚拟机限制的世界，本地方法可以通过本地方法接口来访问虚拟机运行时的数据区，但不止于此，它还可以做任何他想做的事情。比如，它甚至可以直接使用本地处理器中的寄存器，或者直接从本地内存的堆中分配任意数量的内存。总之，它和虚拟机拥有同样的权限（或者说能力）。

本地方法本质上是依赖于实现的，虚拟机实现的设计者可以自由地决定使用怎样的机制来让 Java 程序调用本地方法，任何本地方法接口都会使用某种本地方法栈。当线程调用 Java 方法时，虚拟机会创建一个新的栈帧并压入 Java 栈，然而当它调用的是本地方法时，虚拟机会保持 Java 栈不变，不再在线程的 Java 栈中压入新的帧，虚拟机只是简单地动态连接并直接调用指定的本地方法。我们可以把这个做法看作是虚拟机利用本地方法来动态扩展自己，就如同 Java 虚拟机的实现是按照其中运行的 Java 程序的顺序，调用属于虚拟机内部的另一个（动态连接的）方法。我们知道，当 C 语言编写的程序调用一个 C 函数时，其栈操作都是确定的，首先传递给该函数的参数以

⁷ 即 Garbage Collection。

某个确定的顺序被压入栈，然后它的返回值也以确定的方式被传回给调用者。同样，这就是虚拟机实现本地方法栈的方式，很可能本地方法接口需要回调 Java 虚拟机中的 Java 方法（这也是由设计者决定的），在这种情形下，该线程会保存本地方法栈的状态并进入到另一个 Java 栈。就像其他运行时内存区一样，本地方法栈占用的内存区也不需要是固定大小的，它可以根据需要动态扩展或者收缩，某些 JVM 也允许用户或者程序员指定该内存区的初始大小以及最大、最小值。

注意，在 SUN 的 HOT SPOT 虚拟机中，不区分本地方法栈和虚拟机栈。因此，和虚拟机栈一样，它也会抛出 `StackOverflowError` 和 `OutOfMemoryError`。

2.1.1.4 Java 堆

堆在 JVM 规范里是一种通用性的内存池（也存在于 RAM 中），用于存放所有的 Java 对象。堆是一个运行时数据区，类的对象从中分配空间。这些对象通过 `New` 关键字被建立，它们不需要程序代码来显式地释放。堆是由垃圾回收来负责的，堆的优势是可以动态地分配内存大小，生存周期也不需要事先告诉编译器。由于它是在运行时动态分配内存的，Java 的垃圾收集器会自动收走那些不再使用的数据。但缺点是，由于要在运行时动态分配内存，所以数据存取速度较慢。大多数的虚拟机里，Java 中的对象和数组都存放在堆中。

堆不同于栈的好处是，编译器不需要知道要从堆里分配多少存储区域，也不必知道存储的数据在堆里需要存活多长时间。因此，在堆里分配存储相较于栈来说，有很大的灵活性。当你需要创建一个对象的时候，只需要引用 `New` 关键字写一行简单的代码，当执行这行代码时，会自动在堆里进行存储分配。当然，为这种灵活性必须要付出相应的代价，即用堆进行存储分配比用堆栈进行存储需要更多的时间。

Java 堆区在 JVM 启动的时候即被创建，它只要求逻辑上是连续的，在物理空间上可以是不连续。所有的线程共享 Java 堆，在这里可以划分线程私有的缓冲区（Thread Local Allocation Buffer, TLAB）。

如前所述，Java 堆区是一块用于存储对象实例的内存区，同时也是 GC（Garbage Collection，垃圾收集器）执行垃圾回收的重点区域。正是因为 Java 堆区是 GC 的重点回收区域，那么 GC 极有可能在大内存的使用和频繁进行垃圾回收过程上成为系统性能瓶颈。为了解决这个问题，JVM 的设计者们开始考虑是否一定需要将对象实例存储到 Java 堆区内。基于 OpenJDK⁸深度定制的 TaobaoJVM⁹，其中创新的 GCIH（GC invisible heap）技术实现了 off-heap，即将生命周期较长的 Java 对象从 heap 中移到 heap 之外，并且 GC 不能管理 GCIH 内部的 Java 对象，以此达到降低 GC 的回收频率和提升 GC 的回收效率的目的。除此之外，逃逸分析与栈上分配这样的优化技术同样也是降低 GC 回收频率和提升 GC 回收效率的有效方式。这样一来，Java 堆区就不再是 Java 对象内存分配的唯一选择了。目前主流的垃圾收集算法是按代收集，即按照对象的生存时间分为新生代和老年代。新生代又进一步被划分为 Eden 区、From Survivor 区和 To Survivor 区，主要是为了垃圾回收用途。这里浅显地提一些垃圾回收知识，详细内容请参考第 7 章。

⁸ OpenJDK 做为 GPL 许可（GPL-licensed）的 Java 平台的开源化实现，Sun 正式发布它已经六年有余。从发布那一时刻起，Java 社区的大众们就又开始努力学习，以适应这个新的开源代码基础（code-base）。

⁹ 由 AliJVM 团队发布，是 AliJVM 团队基于 OpenJDK HotSpot VM 发布的国内第一个优化、定制且开源的服务器版 Java 虚拟机。目前已经在淘宝、天猫上线，全部替换了 Oracle 官方 JVM 版本。

逃逸分析英文全名是 `Escape Analysis`。在计算机语言编译器优化原理中，逃逸分析是指分析指针动态范围的方法，它同编译器优化原理的指针分析和外形分析相关联。计算机软件方面，逃逸分析指的是计算机语言编译器语言优化管理中，分析指针动态范围的方法。通俗点讲，如果一个对象的指针被多个方法或线程引用时，那我们可以称这个指针发生了逃逸。Java 语言也有逃逸情况存在，示例代码如代码清单 2-6 所示。

代码清单 2-6 逃逸分析示例代码 `escapeAnalysisClass`

```
public class escapeAnalysisClass{
    public static B b;

    public void globalVariablePointerEscape() { //给全局变量赋值，发生逃逸
        b=new B();
    }

    public B methodPointerEscape() { //方法返回值，发生逃逸
        return new B();
    }

    public void instancePassPointerEscape() {
        methodPointerEscape().printClassName(this); //实例引用发生逃逸
    }
}

class B{
    public void printClassName(G g){
        System.out.println(g.getClass().getName());
    }
}

public class G {
    public static B b;
    public void globalVariablePointerEscape() { //给全局变量赋值，发生逃逸
        b=new B();
    }

    public B methodPointerEscape() { //方法返回值，发生逃逸
        return new B();
    }

    public void instancePassPointerEscape() {
        methodPointerEscape().printClassName(this); //实例引用发生逃逸
    }
}

class B{
    public void printClassName(G g){
        System.out.println(g.getClass().getName());
    }
}
```


代码清单 2-6 所示的这个例子中，一共举了 3 种常见的指针逃逸场景，分别是全局变量赋值、方法返回值、实例引用传递。

逃逸分析研究对于 Java 编译器有什么好处？我们知道 Java 对象总是在堆中被分配的，因此 Java 对象的创建和回收对系统的开销是很大的。Java 语言被批评的一个地方，也是认为 Java 性能慢的一个原因就是 Java 不支持运行时栈分配对象，缺少像 C# 里面的值对象或者 C++ 里面的 struct 结构。JDK6 里的 Swing 内存和性能消耗的瓶颈就是由于发生逃逸所造成的。栈里面只保存了对象的指针，当对象不再被使用后，需要依靠 GC 来遍历引用树并回收内存，如果对象数量较多，将给 GC 带来较大压力，也间接影响了应用的性能。减少临时对象在堆内分配的数量，无疑是最有效的优化方法。

在 Java 应用里普遍存在一种场景，一般是在方法体内，声明了一个局部变量，且该变量在方法执行生命周期内未发生逃逸，因为在方法体内未将引用暴露给外面。按照 JVM 内存分配机制，首先会在堆里创建变量类的实例，然后将返回的对象指针压入调用栈，继续执行。这是 JVM 优化前的方式。

我们可以采用逃逸分析原理对 JVM 进行优化，即针对栈的重新分配方式。首先我们需要分析并且找到未逃逸的变量，将变量类的实例化内存直接在栈里分配（无须进入堆），分配完成后，继续在调用栈内执行，最后线程结束，栈空间被回收，局部变量对象也被回收。通过这种优化方式，与优化前的方案主要区别在栈空间直接作为临时对象的存储介质，从而减少了临时对象在堆内的分配数量。

基于逃逸分析的 JVM 优化原理很简单，但是在应用过程中还是有诸多因素需要被考虑。比如，由于与 JAVA 的动态性有冲突，所以逃逸分析不能在静态编译时进行，必须在 JIT¹⁰里完成。因为你可以在运行时通过动态代理改变一个类的行为，此时，逃逸分析是无法得知类已经变化了。

那么 JIT 怎么通过逃逸分析进行代码优化呢？分析清单 2-7 所示代码。

代码清单 2-7 逃逸分析优化示例代码 my_method

```
public void my_method() {
    V v=new V();
    //use v
    .....
    v=null;
}
```

在这个方法中创建的局部对象被赋给了 v，但是没有返回，没有赋给全局变量等操作，因此这个对象是没有逃逸的，是可以在运行时栈进行分配和销毁的对象。没有发生逃逸的对象由于生命周期都在一个方法体内，因此它们是可以是在运行时栈上分配并销毁。

这样在 JIT 编译 Java 伪代码时，如果能分析出这种代码，那么非逃逸对象其创建和回收就可以在栈上进行，从而能大大提高 Java 的运行性能。

¹⁰ JIT (just in time) 编译器。当 Java 执行 runtime 环境时，每遇到一个新的类，JIT 编译器在此时就会针对这个类进行编译作业。经过编译后的程序，被优化成相当精简的二进制，这种程序的执行速度相当快。

另外，为什么要在逃逸分析之前进行内联分析呢？这是因为往往有些对象在被调用过程中创建并返回给调用过程，调用过程使用完该对象就被销毁了。这种情况下如果将这些方法进行内联，它们就由两个方法体变成一个方法体了，这种原来通过返回传递的对象就变成了方法内的局部对象，就变成了非逃逸对象了，这样这些对象就可以在同一栈上进行分配了。

Java7 开始支持对象的栈分配和逃逸分析机制。这样的机制除能将堆分配对象变成栈分配对象，逃逸分析还有其他两个优化应用。

- 同步消除。我们知道线程同步的代价是相当高的，同步的后果是降低并发性和性能。逃逸分析可以判断出某个对象是否始终只被一个线程访问，如果只被一个线程访问，那么对该对象的同步操作就可以转化成没有同步保护的操作，这样就能大大提高并发程度和性能。
- 矢量替代。逃逸分析方法如果发现对象的内存存储结构不需要连续进行的话，就可以将对象的部分甚至全部都保存在 CPU 寄存器内，这样能大大提高访问速度。

Java 7 完全支持栈式分配对象，JIT 支持逃逸分析优化，此外 Java 7 还默认支持 OpenGL 的加速功能。

堆内垃圾回收

几乎所有的对象和数组都是在堆中分配空间的。Java 堆由年轻代和年老代两个部分组成，年轻代用于存放刚刚产生的对象和年轻的对象，如果对象一直没有被回收，生存得足够长，来年对象就被移入年老代。年轻代又可进一步细分为 eden、survivor space0 和 survivor space1。eden 即对象的出生地，大部分对象刚刚建立时都会被存放在这里。Survivor 空间是存放其中的对象至少经历了一次垃圾回收，并得以幸存下来的。如果在幸存区的对象到了指定年龄仍未被回收，则有机会进入年老代（Tenured）。

对象在内存中的分配方式实例代码如代码清单 2-8 所示。

代码清单 2-8 堆分配示例 TestHeapGC

```
public class TestHeapGC {
    public static void main(String[] args){
        byte[] b1 = new byte[1024*1024/2];
        byte[] b2 = new byte[1024*1024*8];
        b2 = null;
        b2 = new byte[1024*1024*8]; //进行一次年轻代 GC
        System.gc();
    }
}
```

我们针对本例使用 JVM 参数运行程序，读者可以在 Eclipse 这样的 GUI 工具里面设置，也可以在命令行里面配置，具体参数如清单 2-9 所示，我们设置了 60MB 的内存空间作为堆空间。

代码清单 2-9 JVM 参数

```
-XX:+PrintGCDetails -XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=15 -Xms40M
-Xmx40M -Xmn20M
```

清单 2-8 的代码采用 2-9 的配置后运行，GC 输出如清单 2-10 所示。

代码清单 2-10 2-8 代码运行后 GC 输出

```
[GC [DefNew: 9031K->661K(18432K), 0.0022784 secs] 9031K->661K(38912K), 0.0023178
secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
Heap
def new generation    total 18432K, used 9508K [0x34810000, 0x35c10000, 0x35c10000)
eden space 16384K,    54% used [0x34810000, 0x350b3e58, 0x35810000)
  from space 2048K,   32% used [0x35a10000, 0x35ab5490, 0x35c10000)
  to   space 2048K,   0% used [0x35810000, 0x35810000, 0x35a10000)
tenured generation    total 20480K, used 0K [0x35c10000, 0x37010000, 0x37010000)
  the space 20480K,   0% used [0x35c10000, 0x35c10000, 0x35c10200, 0x37010000)
compacting perm gen    total 12288K, used 374K [0x37010000, 0x37c10000, 0x3b010000)
  the space 12288K,   3% used [0x37010000, 0x3706db10, 0x3706dc00, 0x37c10000)
    ro space 10240K,  51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
    rw space 12288K,  55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

从 GC 输出可以看出,在进行多次内存分配的过程中,触发了一次年轻代 GC。在这次 GC 中,原本分配在 eden 段的变量 b1 被移动到 from 空间段(s0)。具体如何读 GC 输出,我们会在第 7 章进一步解释。

我们调整策略,分配的 8MB 内存被分配在 eden 年轻代。再一次运行程序,GC 输出如清单 2-11 所示。

代码清单 2-11 2-8 代码运行后 GC 输出

```
[GC [DefNew: 9031K->661K(18432K), 0.0023186 secs] 9031K->661K(38912K), 0.0023597
secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
[Full GC (System) [Tenured: 0K->8853K(20480K), 0.0179368 secs] 9180K->8853K(38912K),
[Perm : 374K->374K(12288K)], 0.0179893 secs] [Times: user=0.00 sys=0.02, real=0.02 secs]
Heap
def new generation    total 18432K, used 327K [0x34810000, 0x35c10000, 0x35c10000)
eden space 16384K,    2% used [0x34810000, 0x34861f28, 0x35810000)
  from space 2048K,   0% used [0x35a10000, 0x35a10000, 0x35c10000)
  to   space 2048K,   0% used [0x35810000, 0x35810000, 0x35a10000)
tenured generation    total 20480K, used 8853K [0x35c10000, 0x37010000, 0x37010000)
  the space 20480K,  43% used [0x35c10000, 0x364b5458, 0x364b5600, 0x37010000)
compacting perm gen    total 12288K, used 374K [0x37010000, 0x37c10000, 0x3b010000)
  the space 12288K,   3% used [0x37010000, 0x3706db40, 0x3706dc00, 0x37c10000)
    ro space 10240K,  51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
    rw space 12288K,  55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

清单 2-11 的输出显示,在 Full GC 之后,年轻代空间被清空,未被回收的对象全部被移入年老代。

2.1.1.5 方法区

方法区主要保存的信息是类的元数据。方法区与堆空间类似,它也是被 JVM 中所有的线程共享的区域。如图 2-3 方法区组成部分图所示,方法区中最为重要的是类的类型信息、常量池、域信息、方法信息。类型信息包括类的完整名称、父类的完整名称、类型修饰符(Public、Protected、Private)和类型的直接接口类表。

常量池包括类方法、域等信息所引用的常量信息。域信息包括域名称、域类型和域修饰符。方法信息包括方法名称、返回类型、方法参数、方法修饰符、方法字节码、操作数栈和方法栈帧的局部变量区大小以及异常表。方法区是线程间共享的，当两个线程同时需要加载一个类型时，只有一个类会请求 `ClassLoader`¹¹ 加载，另一个线程则会等待。总而言之，方法区内保存的信息大部分来自于 `Class` 文件，是 Java 应用程序运行必不可少的重要数据。

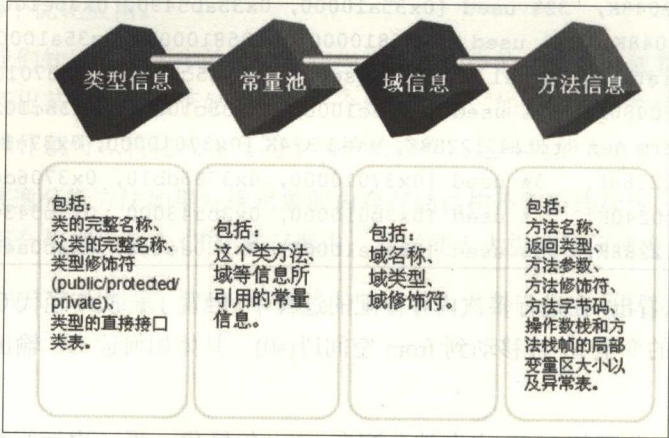


图2-3 方法区组成部分图

在 `HotSpot`¹² 虚拟机中，方法区也被称为永久区，是一块独立于 `Java` 堆的内存空间。虽然被叫作永久区，但是在永久区中的对象同样也是可以被 `GC` 回收的，只是对于 `GC` 的对应策略与 `Java` 堆空间略有不同。

`GC` 针对永久区的回收，通常主要从两个方面分析，一是 `GC` 对永久区常量池的回收，二是永久区对类元数据的回收。

`HotSpot` 虚拟机对常量池的回收策略是很明确的，只要常量池中的常量没有被任何地方引用，就可以被回收。

清单 2-12 所示代码生成大量 `String` 对象，并将其加入常量池中。`String.intern()` 方法的含义是如果常量池中已经存在当前 `String`，则返回池中的对象，如果常量池中不存在当前 `String` 对象，则先将 `String` 加入常量池，并返回池中的对象引用。因此，不停地将 `String` 对象加入常量池会导致永久区饱和，如果 `GC` 不能回收永久区的这些常量数据，那么就会抛出 `OutOfMemoryError` 错误。

代码清单 2-12 回收永久区示例 `permGenGC`

```
public class permGenGC {  
    public static void main(String[] args){  
        for(int i=0;i<Integer.MAX_VALUE;i++){
```

¹¹ 即类加载器，用来加载 `Java` 类到 `Java` 虚拟机中。与普通程序不同的是。`Java` 程序（`class` 文件）并不是本地的可执行程序。当运行 `Java` 程序时，首先运行 `JVM`（`Java` 虚拟机），然后再把 `Java class` 加载到 `JVM` 里头运行，负责加载 `Java class` 的这部分就叫作 `Class Loader`。

¹² `Oracle` 公司收购 `SUN` 公司后整合了原有多种 `JVM` 技术之后推出的一种 `JVM` 实现技术，相对以往的 `JVM`，在性能和扩展能力上都得到了很大的提升，因此它不是一个独立产品，可以理解 `Sun`（`oracle`）实现的 `JVM` 版本的品牌商标。

```
String t = String.valueOf(i).intern();//加入常量池
}
}
}
```

同样的，JVM 设置如清单 2-13 所示。

代码清单 2-13 JVM 设置

```
-XX:PermSize=2M -XX:MaxPermSize=4M -XX:+PrintGCDetails
```

运行 2-12 程序后，GC 输出如清单 2-14 所示。

代码清单 2-14 GC 输出

```
[Full GC [Tenured: 0K->149K(10944K), 0.0177107 secs] 3990K->149K(15872K), [Perm :
4096K->374K(4096K)], 0.0181540 secs] [Times: user=0.02 sys=0.02, real=0.03 secs]
[Full GC [Tenured: 149K->149K(10944K), 0.0165517 secs] 3994K->149K(15936K), [Perm :
4096K->374K(4096K)], 0.0169260 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]
[Full GC [Tenured: 149K->149K(10944K), 0.0166528 secs] 3876K->149K(15936K), [Perm :
4096K->374K(4096K)], 0.0170333 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]
```

从上面 2-14 的输出可以看出，每当常量池饱和时 FULL GC 总能顺利回收常量池数据，确保程序稳定持续进行。

2.1.1.6 缓存

编写高效的程序并不只在于算法的精巧，还应该考虑到计算机内部的组织结构、CPU 微指令的执行、缓存的组织和工作原理等。如果完全没有考虑缓存、微指令实现，作者认为，即使好的算法在实际中不见得有高效率。

缓存的主要作用是暂时保存数据处理结果，并提供下次访问时使用。在很多场合，数据的处理、获取这类操作可能会耗费大量时间，当对数据的请求量很大时，频繁的数据处理会耗尽 CPU 资源。当有其他线程或者客户端需要查询相同的数据资源时，缓存可以帮助我们省略对这些数据的处理流程，直接从缓存中获取处理结果，并立即返回给请求组件，以此提高系统的响应时间。我们这里所指的缓存涵盖 CPU 缓存、内存、磁盘。

早期的 CPU 存储层次只有三层，即 CPU 的寄存器、DRAM 主存和磁盘存储。因为寄存器和主存之间的访问时间开销差距很大，于是设计者在寄存器（一个时钟周期）和主存之间加入了 L1 缓存（2~4 个时钟周期），后来由于 L1 缓存和主存之间的差距，又在主存和 L1 之间加入了 L2 缓存，当然后面还有 L3 缓存。

图 2-4 高速缓存总线结构图所示是高速缓存存储器的典型总线结构。

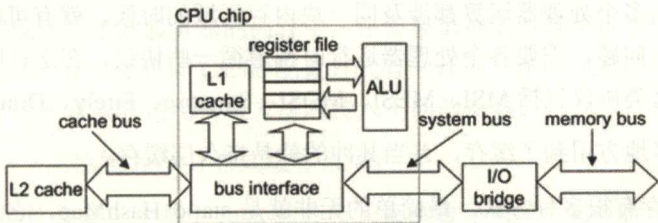


图2-4 高速缓存总线结构

图 2-5 通用缓存的组织结构图清晰地说明了通用缓存的组织结构。

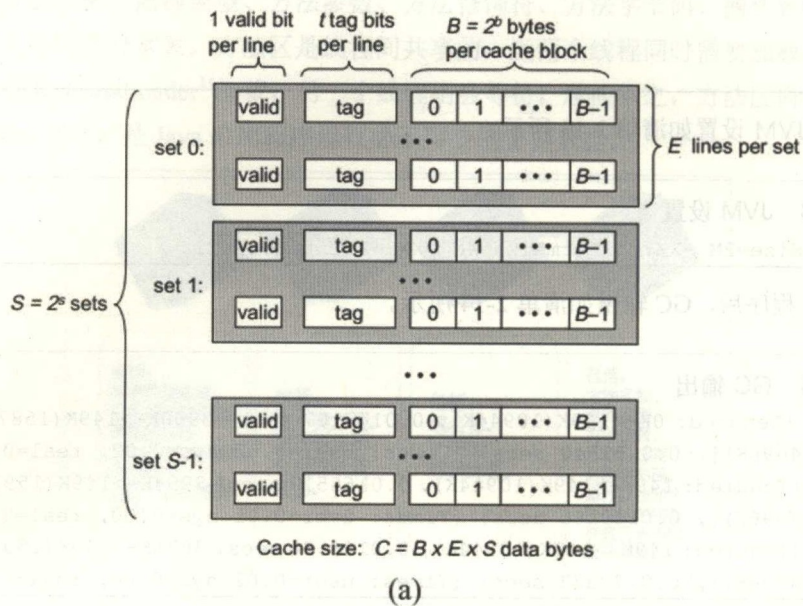


图2-5 通用缓存的组织结构

可以看到，缓存内部是以组的形式存在和组织的。图中的每一块代表一组，每组由一到多行组成（当然图中的是每组有多行）。

计算机 CPU 和内存的交互是最频繁的，内存是我们的高速缓存区，最开始用户直接使用磁盘和 CPU 进行交互，随着 CPU 运转速度越来越快，磁盘渐渐地跟不上 CPU 的读写速度，最后发展到远远跟不上的地步，导致很严重的 I/O 等待成本，进一步导致 CPU 的等待成本。这时我们设计出了内存，用内存来缓存 CPU 计算出来的数据，这一招刚开始也很不错，但是随着 CPU 的不断发展，内存的读写速度也远远跟不上 CPU 的读写速度，更不用说 GPU 的运算速度了。因此，为了解决这一纠纷，CPU 厂商在每颗 CPU 上加入了高速缓存，用来缓解这种症状。

我们知道单核 CPU 的主频不可能无限制的增长，要想较大地提升性能，我们需要涉足多个处理器协同工作。基于高速缓存的存储交互方式很好地解决了处理器与内存之间的矛盾，但也引入了新的问题，即缓存一致性问题。在多处理器系统中，每个处理器有自己的高速缓存，而他们又共享同一块内存，当多个处理器运算都涉及同一块内存区域的时候，就有可能发生缓存不一致的现象。为了解决这一问题，需要各个处理器运行时都遵循一些协议，在运行时需要将这些协议保证数据的一致性。这类协议包括 MSI、MESI、MOSI、Synapse、Firely、DragonProtocol 等。

Java 中也有很多地方用到了缓存，首当其冲的就是持久层缓存。

要实现 Java 缓存有很多种方式，最简单的无非就是 static HashMap，它是基于内存的缓存，一个 Map 可以被用来存储引用对象的缓存。为了区分缓存的数据，我们首先要解决的问题是对象

的有效性以及生命周期。如果这个问题没有很好地解决，很容易就导致内存急剧上升。对象生命周期控制可以采用声明 `SoftReference`、`WeakReference`、`PhantomReference` 这三种对象类型来实现，有别于强引用对象，这三种偏弱引用类型的对象的生命周期与 JVM 挂钩，JVM 内存不足了就会立即被回收，这样能很好地控制 `OutOfMemoryError` 异常。关于对象引用话题的详细内容我们会在第3章深入讨论。

前面已经解释过，为什么要缓存，无非就是节省访问时间，以及大并发量带来的访问上资源的消耗。这个资源包括软资源和硬资源，对象池应该是 Java 程序员最常见的缓存机制。对象池化，是目前很常用的一种系统优化技术。它的核心思想是，如果一个类被频繁请求使用，那么不必每次都生成一个实例，可以将这个类的一些实例保存在一个“池”中，待需要使用的时候直接从池中获取。这个“池”就称为对象池。在实现细节上，它可能是一个数组，一个链表或者任何集合类。对象池的使用非常广泛，例如线程池和数据库连接池。线程池中保存着可以被重用的线程对象，当有任务被提交到线程时，系统并不需要新建线程，而是从池中获得一个可用的线程，执行这个任务。在任务结束后，不需要关闭线程，而将它返回到池中，以便下次继续使用。由于线程的创建和销毁是较为费时的的工作，因此，在线程频繁调度的系统中，线程池可以很好地改善性能。数据库连接池也是一种特殊的对象池，它用于维护数据库连接的集合。当系统需要访问数据库时，不需要重新建立数据库连接，而可以直接从池中获取；在数据库操作完成后，也不关闭数据库连接，而是将连接返回到连接池中。由于数据库连接的创建和销毁是重量级的操作，因此，避免频繁进行这两个操作对改善系统的性能也有积极意义。目前应用较为广泛的数据库连接池组件有 C3P0 和 Proxool。

从实际项目来看，缓存用得好能提高性能，反之会急剧的降低产品的性能。以 Hibernate 为例，理论上来说，Hibernate 性能不如 JDBC，但是如果使用缓存机制较好的话，能更好也未尝不能。Hibernate 缓存最核心的部分是对对象的有效性，缓存的命中率越高意味着性能越高，命中率跟缓存对象的有效性息息相关，如果缓存中对象有效性很差，其性能甚至会低于不用缓存，因为缓存本身就会消耗性能和计算资源，缓存的对象如果很多很快就失效，那效果无疑得不偿失。缓存的深度也有讲究，以 J2EE 项目来说，这个深度是指从页面到数据库的层次结构，显然页面缓存的性能最好，因为调用页面缓存消耗的资源最少，当然现实中是不可能有多页面缓存的。

在开发中大型 Java 软件项目时，很多 Java 架构师都会遇到数据库读写瓶颈，如果你在系统架构时并没有将缓存策略考虑进去，或者并没有选择更优的缓存策略，那么到时候重构起来将会是一个噩梦。目前主流的 5 个常用的 Java 分布式缓存框架，这些缓存框架支持多台服务器的缓存读写功能，可以让你的缓存系统更容易扩展。

■ Ehcache——Java 分布式缓存框架

Ehcache 是一个 Java 实现的开源分布式缓存框架，Ehcache 可以有效地减轻数据库的负载，可以让数据保存在不同服务器的内存中，在需要数据的时候可以快速存取。同时 EhCache 扩展非常简单，官方提供的 Cache 配置方式有好几种。你可以通过声明配置、在 xml 中配置、在程序里配置或者调用构造方法时传入不同的参数。

■ Cacheonix——高性能 Java 分布式缓存系统

Cacheonix 同样也是一个基于 Java 的分布式集群缓存系统，它同样可以帮助你实现分布式缓存的部署。

■ ASimpleCache——轻量级 Android 缓存框架

ASimpleCache 是一款基于 Android 的轻量级缓存框架，它只有一个 Java 文件，ASimpleCache 基本可以缓存常用的 Android 对象，包括普通字符串、JSON 对象、经过序列化的 Java 对象、字节数组等。

■ JBoss Cache——基于事物的 Java 缓存框架

JBoss Cache 是一款基于 Java 的事务处理缓存系统，它的目标是构建一个以 Java 框架为基础的集群解决方案，可以是服务器应用，也可以是 Java SE 应用。

■ Voldemort——基于键-值（key-value）的缓存框架

Voldemort 是一款基于 Java 开发的分布式键-值缓存系统，像 JBoss Cache 一样，Voldemort 同样支持多台服务器之间的缓存同步，以增强系统的可靠性和读取性能。

前面说的都是内存缓存，现在来聊聊硬件缓存。

硬盘缓存（Cache memory）是硬盘控制器上的一块内存芯片，具有极快的存取速度，它是硬盘内部存储和外界接口之间的缓冲器。由于硬盘的内部数据传输速度和外界介面传输速度不同，缓存在其中起到一个缓冲的作用。缓存的大小与速度是直接关系到硬盘的传输速度的重要因素，能够大幅度地提高硬盘整体性能。当硬盘存取零碎数据时需要不断地在硬盘与内存之间交换数据，如果有大缓存，则可以将那些零碎数据暂存在缓存中，减小外系统的负荷，也提高了数据的传输速度。

硬盘的缓存主要起三种作用：一是预读取。当硬盘受到 CPU 指令控制开始读取数据时，硬盘上的控制芯片会控制磁头把正在读取的簇的下一个或者几个簇中的数据读到缓存中（由于硬盘上数据存储时是比较连续的，所以读取命中率较高），当需要读取下一个或者几个簇中的数据的时候，硬盘则不需要再次读取数据，直接把缓存中的数据传输到内存中就可以了，由于缓存的速度远远高于磁头读写的速度，所以能够达到明显改善性能的目的；二是对写入动作进行缓存。当硬盘接到写入数据的指令之后，并不会马上将数据写入到盘片上，而是先暂时存储在缓存里，然后发送一个“数据已写入”的信号给系统，这时系统就会认为数据已经写入，并继续执行下面的工作，而硬盘则在空闲（不进行读取或写入的时候）时再将缓存中的数据写入到盘片上。虽然对于写入数据的性能有一定提升，但也不可避免地带来了安全隐患——如果数据还在缓存里的时候突然掉电，那么这些数据就会丢失。对于这个问题，硬盘厂商们自然也有解决办法：掉电时，磁头会借助惯性将缓存中的数据写入零磁道以外的暂存区域，等到下次启动时再将这些数据写入目的地；第三个作用就是临时存储最近访问过的数据。有时候，某些数据是会经常需要访问的，硬盘内部的缓存会将读取比较频繁的一些数据存储在缓存中，再次读取时就可以直接从缓存中直接传输。

缓存容量的大小不同品牌、不同型号的产品各不相同，早期的硬盘缓存基本都很小，只有几百 KB，已无法满足用户的需求。2MB 和 8MB 缓存是现今主流硬盘所采用，而在服务器或特殊应用领域中还有缓存容量更大的产品，甚至达到了 16MB、64MB 等。大容量的缓存虽然可以在硬盘进行读写工作状态下，让更多的数据存储在缓存中，以提高硬盘的访问速度，但并不意味着缓存越大就越出众。缓存的应用存在一个算法的问题，即便缓存容量很大，而没有一个高效率的算法，那将导致应用中缓存数据的命中率偏低，无法有效发挥出大容量缓存的优势。算法是和缓存容量相辅相成，大容量的缓存需要更为有效率的算法，否则性能会大打折扣，从技术角度上说，高容

量缓存的算法是直接影响到硬盘性能发挥的重要因素。更大容量缓存是未来硬盘发展的必然趋势。

总的来说,无论基于哪种介质上的缓存,缓存都是把双刃剑,一定要根据缓存的特征和业务场景来选择使用才能发挥其最大优势。

2.1.1.7 缓冲区

缓冲区是一块特定的内存区域。开辟缓冲区的目的是通过缓解应用程序上下层之间的性能差异,提高系统的性能。在日常生活中,缓冲的一个典型应用是漏斗,如图2-6所示。

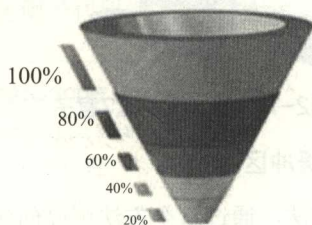


图2-6 模拟缓冲区

缓冲可以协调上层组件和下层组件的性能差,当上层组件性能优于下层组件时,可以有效减少上层组件对下层组件的等待时间。基于这样的结构,上层应用组件不需要等待下层组件真实地接受全部数据,即可返回操作,加快了上层组件的处理速度,从而提升系统整体性能。

JDK 里的 `BufferedWriter` 就是一个缓冲区用法,一般来说,缓冲区不宜过小,过小的缓冲区无法起到真正的缓冲作用,缓冲区也不宜过大,过大的缓冲区会浪费系统内存,增加 GC 负担。尽量在 I/O 组件内加入缓冲区,可以提高性能。

如果将同步 I/O 方式下的数据传输比作数据传输的零星方式(这里的零星是指在数据传输的过程中是以零星的字节方式进行的),那么就可以将非阻塞 I/O 方式下的数据传输比作数据传输的集装箱方式(在字节和低层数据传输之间,多了一层缓冲区,因此,可以将缓冲区看作是装载字节的集装箱)。大家可以想象,如果我们要运送比较少的货物,用集装箱好像有点不太合算,而如果要运送上百吨的货物,用集装箱来运送的成本会更低。在数据传输过程中也是一样,如果数据量很小时,使用同步 I/O 方式会更适合,如果数据量很大时(一般以 GB 为单位),使用非阻塞 I/O 方式的效率会更高。因此,从理论上说,数据量越大,使用非阻塞 I/O 方式的单位成本就会越低。产生这种结果的原因和缓冲区的一些特性有着直接的关系。

如图2-7所示,Java 提供了7个基本的缓冲区,分别由7个类来管理,它们都可以在 `java.nio` 包中找到。这7个类分别是 `ByteBuffer`、`ShortBuffer`、`IntBuffer`、`CharBuffer`、`FloatBuffer`、`DoubleBuffer`、`LongBuffer`。

这七个类中的方法类似,只是它们的返回值或参数和相应的简单类型相对应,如 `ByteBuffer` 类的 `get` 方法返回了 `byte` 类型的数据,而 `put` 方法需要一个 `byte` 类型的参数。在 `CharBuffer` 类中的 `get` 和 `put` 方法返回和传递的数据类型就是 `char`。这7个类都没有 `public` 构造方法,因此,它们不能通过 `new` 来创建相应的对象实例。这些类都可以通过两种方式来创建相应的对象实例。

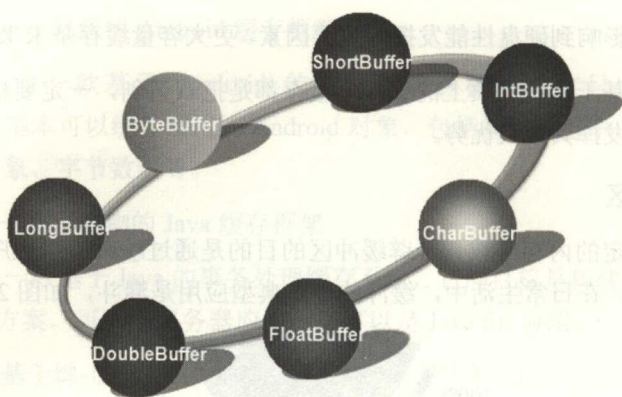


图2-7 Java缓冲区7君子

1. 通过静态方法 allocate 来创建缓冲区

这七类都有一个静态的 allocate 方法，通过这个方法可以创建有最大容量限制的缓冲区对象。allocate 的定义如下：

- ByteBuffer 类中的 allocate 方法——`public static ByteBuffer allocate(int capacity)`。
- IntBuffer 类中的 allocate 方法——`public static IntBuffer allocate(int capacity)`。

其他五个缓冲区类中的 allocate 方法定义和上面的定义类似，只是返回值的类型是相应的缓冲区类。

allocate 方法有一个参数 capacity，用来指定缓冲区容量的最大值。capacity 的不能小于 0，否则会抛出一个 `IllegalArgumentException` 异常。使用 allocate 来创建缓冲区，并不是一下子就分配给缓冲区 capacity 大小的空间，而是根据缓冲区中存储数据的情况来动态分配缓冲区的大小（实际上，在低层 Java 采用了数据结构中的堆来管理缓冲区的大小），因此，这个 capacity 可以是一个很大的值，如 $1024 \times 1024 (1M)$ 。allocate 的使用方法如下：

```
ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
```

```
IntBuffer intBuffer = IntBuffer.allocate(1024);
```

在使用 allocate 创建缓冲区时应用注意，capacity 的含义随着缓冲区的不同而不同。如创建字节缓冲区时，capacity 指的是字节数。而在创建整型（int）缓冲区时，capacity 指的是 int 型值的数目，如果转换成字数，capacity 的值应该乘 4。如上面代码中的 intBuffer 缓冲区最大可容纳的字节数是 $1024 \times 4 = 4096$ 个。

2. 通过静态方法 wrap 来创建缓冲区。

使用 allocate 方法可以创建一个空的缓冲区。而 wrap 方法可以利用已经存在的数据来创建缓冲区。wrap 方法可以将数组直接转换成相应类型的缓冲区。wrap 方法有两种重载形式，它们的定义如下。

ByteBuffer 类中的 wrap 方法：

```
public static ByteBuffer wrap(byte[] array)
```

```
public static ByteBuffer wrap(byte[] array, int offset, int length)
```


IntBuffer 类中的 wrap 方法:

```
public static IntBuffer wrap(byte[] array)
public static IntBuffer wrap(byte[] array, int offset, int length)
```

其他五个缓冲区类中的 wrap 方法定义和上面的定义类似,只是返回值的类型是相应的缓冲区类。

应用程序和 I/O 设备之间存在一个缓冲区,一般流是没有缓冲区的,但是如果存在缓冲区,就会发现很大的问题。一个缓冲区例子代码清单 2-15 所示。

代码清单 2-15 无缓冲区示例 testCache

```
import java.io.BufferedOutputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;

public class testCache {
    public static void main(String[] args) throws Exception{
        DataOutputStream out = new DataOutputStream(
            new BufferedOutputStream(
                new FileOutputStream("1.txt")));
        out.writeChars("hello");
        FileInputStream in = new FileInputStream("1.txt");
        int len = in.available();
        byte[] b = new byte[len];
        int actlen = in.read(b);
        String str = new String(b);
        System.out.println(str);
    }
}
```

为了确保问题不发生,清单 2-15 所示代码中使用了 BufferedOutputStream,手动构造出了一个缓冲区。因为如果没有缓冲区,应用程序每次 I/O 都要和设备进行通信,效率很低,因此缓冲区为了提高效率,当写入设备时,先写入缓冲区,等到缓冲区有足够多的数据时,就整体写入设备。这就是问题所在,上个例子中,当我们写入 hello 时,由于 hello 占用空间很小,所以暂时存放在缓冲区中,后来输入流想要从文件中读取,但是由于文件中没有字节,所以不能读取 hello。这里,解决方法很简单,只要调用 out.flush() 或者 out.close()即可,这是把缓冲区的数据手动写入文件。代码如清单 2-16 所示。

代码清单 2-16 增加缓冲区后示例 testCache

```
import java.io.BufferedOutputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
```



```

public class testCache {
    public static void main(String[] args) throws Exception{
        DataOutputStream out = new DataOutputStream(
            new BufferedOutputStream(
                new FileOutputStream("1.txt")));
        out.writeChars("hello");
        out.close();//inserted
        FileInputStream in = new FileInputStream("1.txt");
        int len = in.available();
        byte[] b = new byte[len];
        int actlen = in.read(b);
        String str = new String(b);
        System.out.println(str);
    }
}

```

缓冲区溢出是一种常见的缓冲区使用不当导致的缺陷。具体是指当计算机程序向缓冲区内填充的数据位数超过了缓冲区本身的容量，溢出的数据覆盖在合法数据上。理想情况是，程序检查数据长度并且不允许输入超过缓冲区长度的字符串，但是绝大多数程序都会假设数据长度总是与所分配的存储空间相匹配，这就为缓冲区溢出埋下隐患。操作系统所使用的缓冲区又被称为堆栈，在各个操作进程之间，指令被临时存储在堆栈当中，堆栈也会出现缓冲区溢出。当一个超长的数据进入到缓冲区时，超出部分就会被写入其他缓冲区，其他缓冲区存放的可能是数据、下一条指令的指针，或者是其他程序的输出内容，这些内容都被覆盖或者破坏掉。可见一小部分数据或者一套指令的溢出就可能导致一个程序或者操作系统崩溃。JVM 由于内置的安全措施较好，所以缓冲区溢出问题不是很严重。

综上所述，缓冲区还是很有用的一套“漏斗”机制，读者可以考虑在自己的应用中适当使用。

2.1.2 GPU/CPU

所有的软件系统都离不开计算，也就离不开对 CPU/GPU 的依赖，这一节具体聊聊 CPU/GPU 的发展前景。

2.1.2.1 CPU 发展前景分析

进入新世纪以来，CPU 进入了更高速发展的时代，以往可望而不可即的 1Ghz 大关被轻松突破了，在市场分布方面，仍然是 Intel 跟 AMD 公司在两雄争霸，它们分别推出了 Pentium4、Tualatin 核心 Pentium III 和 Celeron, Tunderbird 核心 Athlon、AthlonXP 和 Duron 等处理器，竞争日益激烈。

根据最新的报道，英特尔将在 2015 年推出 18 核心的 Broadwell 处理器。同时另一份报告指出，未来 Broadwell 处理器将有望被使用在平板电脑上。

此外，英特尔还计划推出 8 核至 10 核的高性能桌面和服务处理器。现在该公司在市面上的最新的处理器架构为 Haswell，Broadwell 处理器将在 2016 年上半年上市。目前英特尔处理器最高的核心数为 12 个。在软件能够支持多核的前提下，显然更多的核心能为设备带来更好的性能。另据 CPU World 报道，在发展多核的同时，英特尔在节能的方向上也取得了进展，未来 Broadwell 处理器的功耗将可能低至 4.5 瓦，这将让这款处理器更广泛地被运用到平板电脑和二合一变形本上，

如今移动计算正是英特尔重点关注的领域。

表 2-1 列出了与 CPU 相关的一些术语，这些术语在 Java 编程中都会具体涉及，先在这里列给大家。

表 2-1 CPU 相关术语

术语	英文单词	术语描述
内存屏障	Memory barriers	是一组处理器指令，用于实现对内存操作的顺序限制
缓冲行	Cache line	缓存中可以分配的最小存储单位。处理器填写缓存线时会加载整个缓存线，需要使用多个主内存读周期
原子操作	Atomic operations	不可中断的一个或一系列操作
缓存行填充	Cache line fill	当处理器识别到从内存中读取操作数是可缓存的，处理器读取整个缓存行到适当的缓存（L1、L2、L3 的或所有）
缓存命中	Cache hit	如果进行高速缓存行填充才做的内存位置仍然是下次处理器访问的地址时，处理器从缓存中读取操作数，而不是从内存读取
写命中	Write hit	当处理器将操作数写回到一个内存缓存的区域时，它首先会检查这个缓存的内存地址是否在缓存行中，如果存在一个有效的缓存行，则处理器将这个操作数写回到缓存，而不是写回到内存，这个操作被称为写命中
写缺失	Write misses the cache	一个有效的缓存行被写入到不存在的内存区域
缓存行	Cache Line	缓存的最小操作单位
比较并交换	Compare and Swap	CAS 操作需要输入两个数值，一个旧值和一个新值，在操作期间先比较旧值有没有发生变化，如果没有发生变化，才交换成新值，发生了变化则不交换
CPU 流水线	CPU pipeline	CPU 流水线的工作方式就像工业生产商的装配流水线，在 CPU 中由 5-6 个不同功能的电路单元组成一条指令处理流水线，然后将一条 X86 指令分成 5-6 步后再由这些电路单元分别执行，这样就能实现在一个 CPU 时钟周期完成一条指令，因此提高 CPU 的运算速度
内存顺序冲突	Memory order violation	内存顺序冲突一般是由假共享引起的，假共享是指多个 CPU 同时修改同一个缓存行的不同部分而引起其中一个 CPU 的操作无效，当出现这个内存顺序冲突时，CPU 必须清空流水线

2.1.2.2 GPU 发展前景分析

GPU（GraphicsProcessing Unit）是图形处理器的简称，这个概念是由 NVIDIA 公司在发布 GeForce256 绘图处理芯片时首先提出。

GPU 使显卡减少了对 CPU 的依赖，并分担了部分原本是由 CPU 所担当的工作，尤其是在进行 3D 图形处理时，功效更加明显。GPU 所采用的核心技术有硬件坐标转换与光源、立方环境材质贴图和顶点混合、纹理压缩和凹凸映射贴图、双重纹理四像素 256 位渲染引擎等。

GPU 在处理能力和存储器带宽上相对于 CPU 有明显优势，在成本和功耗上也不需要付出太大代价。由于图形渲染的高度并行性，使得 GPU 可以通过增加并行处理单元和存储器控制单元的方

式提高处理能力和存储器带宽。GPU 设计者将更多的晶体管用作执行单元，而不是像 CPU 那样用作复杂的控制单元和缓存并以此来提高少量执行单元的执行效率。图 2-8 对 CPU 与 GPU 中的逻辑架构进行了对比。

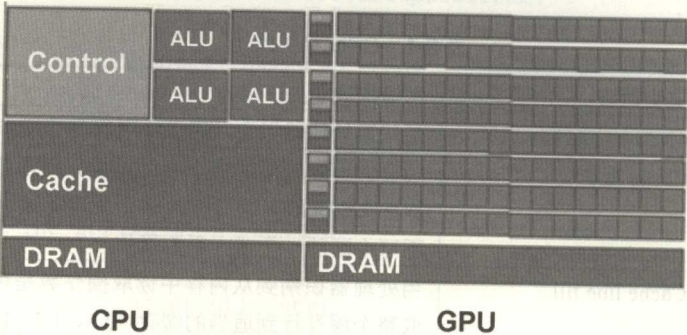


图2-8 CPU对比GPU

CPU 的整数计算、分支、逻辑判断和浮点运算分别由不同的运算单元执行，此外还有一个浮点加速器。因此，CPU 面对不同类型的计算任务会有不同的性能表现。而 GPU 是由同一个运算单元执行整数和浮点计算，因此，GPU 的整型计算能力与其浮点能力相似。

总的来说，GPU 较 CPU 来说，拥有以下这些优势。

■ 更高的内存带宽 (MemoryBandwidth)

GPU 运算相对于 CPU 还有一项巨大的优势，那就是其内存子系统，也就是 GPU 上的显存。当前桌面级顶级产品 3 通道 DDR3-1333 的峰值是 32GB/S，实测中由于诸多因素带宽在 20GB/S 上下浮动。AMDHD 4870512MB 使用了带宽超高的 GDDR5 显存，内存总线数据传输率为 3.6T/s 或者说 107GB/s 的总线带宽。存储器的超高带宽让巨大的浮点运算能力得以稳定吞吐，也为数据密集型任务的高效运行提供了保障。

还有，从 GTX200 和 HD4870 系列 GPU 开始，AMD 和 NVIDIA 两大厂商都开始提供对双精度运算的支持，这正是不少应用领域的科学计算都需要的。NVIDIA 公司最新的 Fermi 架构更是将全局 ECC (ErrorChecking and Correcting)、可读写缓存、分支预测等技术引入到 GPU 的设计中，明确了将 GPU 作为通用计算核心的方向。

■ 延迟与带宽

GPU：高显存带宽和很强的处理能力提供了很大的数据吞吐量，缓存不检查数据一致性，直接访问显存延时可达数百乃至上千时钟周期。

CPU：通过大的缓存保证线程访问内存的低延迟，但内存带宽小，执行单元太少，数据吞吐量小，需要硬件机制保证缓存命中率和数据一致性。

尽管 GPU 计算已经开始崭露头角，但 GPU 并不能完全替代 X86 解决方案。很多操作系统、软件以及部分代码现在还不能运行在 GPU 上，所谓的 GPU+CPU 异构超级计算机也并不是完全基于 GPU 进行计算。一般而言适合 GPU 运算的应用特性包括运算密集型、高度并行、控制简单、分多个阶段执行这些方面。

前面说过，GPU 计算的优势是大量内核的并行计算，瓶颈往往是 I/O 带宽，因此适用于计算

密集型的计算任务。目前的 GPU 开发环境很多，主要有如下 5 种。

- (1) CG (C for Graphics) 是为 GPU 编程设计的高级绘制语言，由 NVIDIA 和微软联合开发，微软版本叫 HLSL，CG 是 NVIDIA 版本。Cg 极力保留 C 语言的大部分语义，并让开发者从硬件细节中解脱出来，Cg 同时也有一个高级语言的其他好处，如代码的易重用性，可读性得到提高，编译器代码优。
- (2) CUDA (Compute Unified Device Architecture，统一计算架构) 是由 NVIDIA 所推出的一种集成技术，是该公司对于 GPGPU 的正式名称。通过这个技术，用户可利用 NVIDIA 的 GeForce8 以后的 GPU 和较新的 QuadroGPU 进行计算。亦是首次可以利用 GPU 作为 C-编译器的开发环境。NVIDIA 营销的时候，往往将编译器与架构混合推广，造成混乱。实际上，CUDA 架构可以兼容 OpenCL 或者自家的 C-编译器。无论是 CUDAC-语言或是 OpenCL，指令最终都会被驱动程序转换成 PTX 代码，交由显示核心计算。
- (3) ATISStream 是 AMD 针对旗下图形处理器 (GPU) 所推出的通用并行计算技术。利用这种技术可以充分发挥 AMDGPU 的并行运算能力，用于对软件进行加速或进行大型的科学运算，同时用以对抗竞争对手的 nVIDIACUDA 技术。与 CUDA 技术是基于自身的私有标准不同，ATISStream 技术基于开放性的 OpenCL 标准。
- (4) OpenCL (Open Computing Language，开放计算语言) 是一个为异构平台编写程序的框架，此异构平台可由 CPU，GPU 或其他类型的处理器组成。OpenCL 由一门用于编写 kernels (在 OpenCL 设备上运行的函数) 的语言 (基于 C99) 和一组用于定义并控制平台的 API 组成。OpenCL 提供了基于任务分区和数据分区的并行计算机制。
- (5) OpenCL 类似于另外两个开放的工业标准 OpenGL 和 OpenAL，这两个标准分别用于三维图形和计算机音频方面。OpenCL 扩展了 GPU 用于图形生成之外的能力。OpenCL 由非盈利性技术组织 KhronosGroup 掌管。

总的来说，CUDA 仅能用于 NVIDIA 的产品，发展相对成熟，效率高，拥有丰富的文档资源。OpenCL 的开放标准，抽象层次较低，较多对硬件的直接操作，代码需要根据不同硬件优化。ATISStream 在硬件上已经有了基础，但只有低层次汇编才能使用所有的硬件资源。高层次的 brook 是基于上一代 GPU 的，缺乏良好的编程模型。CG 是优秀的图形学开发环境，但不适于 GPU 通用计算开发。

2015 年 NVIDIA 公司推出的 Titan X 的 TDP 功耗为 250W (该卡仅预留了一个 8pin 和一个 6pin 供电接口，总功耗不会超过 300W)，拥有 6 组 GPC (Graphics Processing Clusters，图形处理集群)，每一集群都拥有 4 组 SMM 单元，也就是说 Titan X 总共拥有 24 组 SMM 单元，3072 个 CUDA 核心。

多年来，有很多将 Hadoop 或 MapReduce 应用到 GPU 的科研项目。Mars 可能是第一个成功的 GPU 的 MapReduce 框架。采用 Mars 技术，分析 WEB 数据 (搜索和日志) 和处理 WEB 文档的性能提高了 1.5~1.6 倍。根据 Mars 的基本原理，很多科研机构都开发了类似的工具，提高自己数据密集型系统的性能。相关案例包括分子动力学、数学建模 (如 Monte Carlo)、基于块的 矩阵乘法、财务分析、图像处理等。

还有针对网格计算的 BOING 系统，它是一个快速发展、志愿者驱动的中间件系统。尽管没有

使用 Hadoop, BOINC 已经成为许多科研项目加速的基础。例如, GPUGRID 是一个基于 BOINC 的 GPU 和分布式计算的项目,它通过执行分子模拟,帮助我们了解蛋白质在健康和疾病情况下的不同作用。多数关于医药、物理、数学、生物等的 BOINC 项目也可以使用 Hadoop+GPU 技术。

因此,使用 GPU 加速并行计算系统的需求是存在的。这些机构会投资 GPU 的超级计算机或开发自己的解决方案。硬件厂商,如 Cray,已经发布了配置 GPU 和预装了 Hadoop 的机器。Amazon 也推出了 EMR(Amazon Elastic MapReduce),用户可以在其配置了 GPU 的服务器上使用 Hadoop。

数据处理过程中, HDD、DRAM、CPU 和 GPU 必然会有数据交换。图 2-9 显示了 CPU 和 GPU 共同执行计算时,数据的传输。

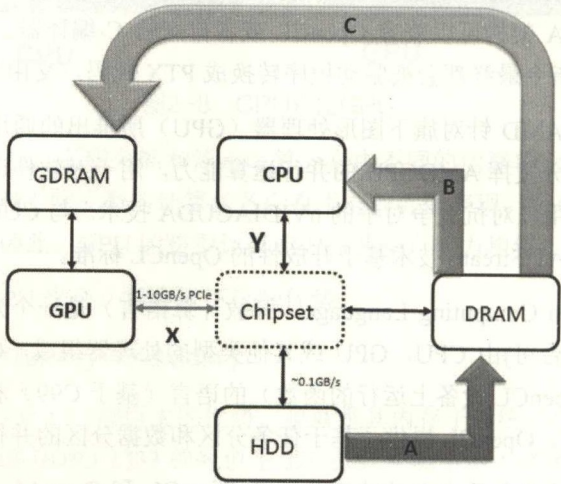


图2-9 共同计算流程图

- 箭头 A: 数据从 HDD 传输到 DRAM (CPU+GPU 计算的初始步骤)
- 箭头 B: CPU 处理数据 (数据流: DRAM->chipset->CPU)
- 箭头 C: GPU 处理数据 (数据流: DRAM->chipset->CPU->chipset->GPU->GDRAM->GPU)

OpenJDK 发起的另外一个非常有潜力的项目 Sumatra,旨在通过 GPU 来大幅提高 Java 性能。可以将运行 Java 程序的部分计算工作从 CPU 移动到 GPU。此想法将通过 Hotspot JVM 来实现, Hotspot JVM 具有先进的代码性能运行时分析功能,开发者将可以查看生成的 GPU 代码、围绕代码的垃圾收集等。该项目的目的是提升性能,但是并未影响到编译时间、内存消耗和生成代码的质量等。

■ CPU 与 GPU 结合

CPU/GPU 异构混合并行系统以其强劲计算能力、高性价比和低能耗等特点成为新型高性能计算平台,但其复杂体系结构为并行计算研究提出了巨大挑战。CPU/GPU 协同并行计算属于新兴研究领域,是一个开放的课题。

采用通用多核微处理器与定制加速协处理器相结合的异构混合体系结构成为构造千万亿次计算机系统的一种可行途径。甚至有专家预言,今后的高性能计算平台将会成为以异构混合体系结构为主的格局。在众多异构混合平台中,基于 CPU/GPU 异构协同的计算平台具有很大的发展潜力。

正由于 GPU 所具有的强劲计算能力、高性能/价格比和高性能/能耗比,在当今追求绿色高性能计算的时代, GPU 的计算优势受到越来越多的关注。除专业图形应用外, GPU 已用于大量的通用计算问题,并形成了 GPU 通用计算研究领域。

GPU 和 CPU 在设计思路存在很大差异, CPU 为优化串行代码而设计,将大量的晶体管作为控制和缓存等非计算功能,注重低延迟地快速实现某个操作; GPU 则将大量的晶体管用作 ALU 计算单元,适合高计算强度(计算/访存比)的应用。在协同并行计算时, CPU 和 GPU 应各取所长,快速、高效协同地完成高性能计算任务。另外,除管理 GPU 计算任务外, CPU 也应当承担一部分科学计算任务。需要充分挖掘 CPU 和 GPU 的计算潜能,使其达到高效协同的计算效果。新型异构混合体系结构对大规模并行算法研究提出了新的挑战,迫切需要深入研究与该体系结构相适应的并行算法。针对 CPU/GPU 异构混合体系结构的高性能计算平台,研究相应的协同并行计算技术,设计并实现大型科学及工程计算问题的新型并行算法,具有重大的理论和实际意义。

在 CPU/GPU 异构混合平台中, CPU 和 GPU 具有不同的硬件特点和计算方式,因此基于异构混合平台进行并行算法设计时,必须密切结合其底层硬件特点,使算法充分利用混合系统中各类型处理器的性能优势。鉴于 GPU 研究属于新兴领域,目前大部分算法研究工作是已有算法向异构混合平台的移植,针对该平台的全新算法较少。CPU 和 GPU 都存在存储墙问题, CPU 主要通过多层次存储结构来缓解该问题,而 GPU 则使用硬件多线程技术来隐藏高开销的访存延迟。面向异构混合系统的高效并行算法应具有以下特点。

- 异构感知的:根据底层硬件特点设计算法,使体系结构—算法组合发挥出最大性能。
- 计算强度高:高计算强度是并行程序高计算效率的普遍要求,对 GPU 尤其重要,否则 GPU 的高浮点计算性能优势根本得不到发挥。
- CPU 与 GPU 交互开销小:包括数据传输开销及同步开销。
- CPU 与 GPU 间交互是协同并行计算不可避免的:应通过优化算法来减少数据传输次数和数据量以及同步开销。

2.1.3 硬盘

软件系统架构离不开硬盘,这是由于大多数软件都需要有存储介质支撑,不然软件运行过程中产生出来的数据没有地方保存,所以我们也需要一定程度地了解硬盘相关知识。

硬盘对于计算机来说,就像人们要吸收和存储大量信息,离不开头脑一样。信息越来越多,硬盘容量也在飞速增长,一般来说,60%的年度容量增长速度对应的是40%的价格减少。

SATA (Serial ATA) 口的硬盘又叫串口硬盘,是未来 PC 机硬盘的趋势,现已基本取代了传统的 PATA 硬盘。SATA 的全称是 Serial Advanced Technology Attachment, Intel、APT、Dell、IBM、希捷、迈拓这几大厂商组成的 Serial ATA 委员会正式确立了 Serial ATA 1.0 规范,2002 年,虽然串行 ATA 的相关设备还未正式上市,但 Serial ATA 委员会已抢先确立了 Serial ATA 2.0 规范。Serial ATA 采用串行连接方式,串行 ATA 总线使用嵌入式时钟信号,具备了更强的纠错能力,与以往相比其最大的区别在于能对传输指令(不仅仅是数据)进行检查,如果发现错误会自动矫正,这在很大程度上提高了数据传输的可靠性。串行接口还具有结构简单、支持热插拔的优点。

固态硬盘 (Solid State Drives),简称固盘,固态硬盘用固态电子存储芯片阵列而制成的硬盘,

由控制单元和存储单元（FLASH 芯片、DRAM 芯片）组成。固态硬盘在接口的规范和定义、功能及使用方法上与普通硬盘的完全相同，在产品外形和尺寸上也完全与普通硬盘一致。被广泛应用于军事、车载、工控、视频监控、网络监控、网络终端、电力、医疗、航空、导航设备等领域。

闪存（Flash Memory）是一种长寿命的非易失性（在断电情况下仍能保持所存储的数据信息）的存储器，数据删除不是以单个的字节为单位而是以固定的区块为单位（注意：NOR Flash 为字节存储），区块大小一般为 256KB 到 20MB。闪存是电子可擦除只读存储器（EEPROM）的变种，闪存与 EEPROM 不同的是，EEPROM 能在字节水平上进行删除和重写而不是整个芯片擦写，而闪存的大部分芯片需要块擦除。由于其断电时仍能保存数据，闪存通常被用来保存设置信息，如在电脑的 BIOS（基本程序）、PDA（个人数字助理）、数码相机中保存资料等。

硬盘发展至今已经有 50 余年的历史，在这几十年的历程里，硬盘的体积越来越小而容量则越来越大，硬盘的转速与接口也在与时俱进。第一款硬盘面世的时候，它有两个冰箱那么宽，内部安装了 50 个直径两英尺的磁盘，重量约 1 吨，而现在微硬盘、CF 硬盘仅仅才硬币大小，这种变化真是太惊人了。

由于硬盘的 I/O 速度一直是造成软件系统瓶颈的因素之一，所以很多时候我们在设计高吞吐量系统的时候会避开硬盘，比如通过消息队列的方式加快数据处理流程的流动。比如异构架构的服务器可能会给每颗处理器（包括 CPU 和 GPU 的处理器，比如说英伟达公司的 K1、X1）配置闪存，通过闪存来存储需要的操作系统、软件，通过从闪存加载的方式避免使用硬盘，又通过内存来保证系统的运行，但是这样不能解决数据持久化需求。

针对海量存储的需求领域，业界有两点共识很重要。

- （1）海量存储不可能再以传统的块，文件系统，应用的访问方式进行建设，未来的主流很可能对象存储；
- （2）传统的服务器或存储系统，增加了存储成本和管理难度，未来的应用将向更细颗粒度的存储组件发展；

IP 硬盘就是在这样的背景下产生了，也是在这样的背景下与海量存储联系在了一起。

首先，对于主机+NIC+IP 硬盘与传统的主机+SAS HBA+传统硬盘相比较成本优势明显，且维护管理方便；其次，IP 硬盘的访问方式基于 KEY/VALUE，某种意义上来说，就是一种对象存储，在支持大规模并行访问上，显然是有着得天独厚的优势；再则，每个 IP 硬盘即是一个最细粒度的存储组件，扩容非常方便；所以，相比较传统的 SAS 盘，IP 硬盘在成本，性能及可靠性上都占有优势，且 IP 硬盘更容量管理，对于对象存储天然的支持的特性，使得 IP 硬盘将是未来发展的一个重要方向。

现在市场上的 IP 硬盘产品主要是希捷的 IP 硬盘，希捷推出了 kinetic 系统加 IP 硬盘的整体解决方案，这个系统基于 KEY/VALUE 的存储方式，存储单位为 1MB 大小，并以 lib 库+IP 硬盘的形式提供出来；大数据技术部云存储组目前针对希捷推出的这套 Kinetic 系统的 IP 硬盘，分别测试过 Python、C、Java 版本，单盘异步写可以达到 50MB/S 的性能，基本能满足我们的性能要求；其整体解决方案给我们的存储架构设计方面带来了一些启发，我们构想，新一代的存储产品，会是基于对象的存储，由现在的前端+元数据服务器+数据服务器，变更为前端+元数据服务器+IP 硬盘，节省开发存储服务器的时间成本，但会带来大量索引 I/O 的管理难度，后期的存储设计会集

中在如何处理海量的元数据索引上,对于可靠性的保护也从后端转移到前端,对于网络 raid 的研究就显得比较迫切;目前正积极同希捷 IP 硬盘部门进行沟通,深入了解 IP 硬盘的特性以及设计思路,从技术、市场方面进行分析,投入人员进行预研,深入软件定义存储。

总的来说,IP 硬盘的设计对于冷数据,也即低负载、以带宽吞吐量大块连续 I/O 为主的场景有着很好的支持。结合现在比较流行的 Erasure code 技术,首先对数据进行分片,然后再保存到各个 IP 硬盘中,这样为更高的数据可靠性和存储利用率提供了可能。

2.1.4 网络架构

现代软件系统,特别是集群式、分布式软件架构方式的系统,系统内部一定牵扯到网络带宽问题,这样也就涉及到了网络设计、架构技术。比如国内腾讯、百度、阿里三甲互联网公司,为满足用户访问,平均每两周就有 1000 台以上服务器上线,这样的上线速度和数量,对整个数据中心的自动化运维提出了极高的要求,基础的网络同样需要适应这种需求。

软件系统内部实现了中心节点管理服务为例,管理各个具体执行某样业务的计算节点,那么中心节点与计算节点之间,计算节点与计算节点之间,网络负载问题是我们必须要考虑的。无论系统采用的 Master-Slave¹³式架构,或是去中心化的 ZooKeeper¹⁴式通信方式架构,我们都不能无视网络风暴存在的可能性,所以这里具体和大家聊聊网络相关知识。

当前最火的名词可能数据中心得算一个吧,随着云计算的日益火热,数据中心概念也被热炒。新一代数据中心解决方案,目标是在以太网和 IP 技术的基础上,实现数据中心基础网络架构的统一,以及安全策略的统一部署和数据中心资源的统一管理,以帮助用户简化传统数据中心的基础架构、加固核心数据的保护、优化数据中心的应用性能。

网络是数据中心的一个重要基础设施,通常可以分为两大部分:前端计算网络、后端存储网络。后端的存储网络目前主要是 FC 网络,不过随着 IP 技术的发展,采用 FCoE 技术融合前后端网络成为一种趋势。

传统的机房前端接入网络主要由大量的二层接入设备及少量的三层设备组成,即分为接入层、汇聚层、核心层。传统的网络模型在很长一段时间内,支撑了各种类型的数据中心,但随着互联网的发展以及云计算的崛起,已经不能再有效地支撑数据中心了。万兆以太网从起步到目前逐渐成为应用主流,延续了以太网技术发展的主基调,凭借其技术优势,替代其他网络接入技术,成为高性能网络的不二选择。

传统的数据中心内,服务器主要用于对外提供业务访问,不同的业务通过安全分区及 VLAN 隔离。一个分区通常集中了该业务所需的计算、网络及存储资源,不同的分区之间或者禁止互访,或者经由核心交换通过三层网络交互,数据中心的网络流量大部分集中于南北向。此种设计下,不同分区间计算资源无法共享,资源利用率低下的问题越来越突出。通过虚拟化技术、云计算管理技术等,将各个分区间的资源进行池化,实现数据中线内资源的有效利用,虚拟机迁移、数据

¹³ 一种经典的命令模式, Master 和 Slave 之间通过相互发送命令 (Command) 实现交互。

¹⁴ ZooKeeper 是一个分布式的,开放源码的分布式应用程序协调服务,是 Google 的 Chubby 一个开源的实现,是 Hadoop 和 Hbase 的重要组件。它是一个为分布式应用提供一致性服务的软件,提供的功能包括:配置维护、名字服务、分布式同步、组服务等。

同步、数据备份、协同计算等在数据中心内开始实现部署，数据中心内部东西向流量开始大幅度增加。

随着虚拟化数据中心的扩大，以及云化管理的深入，物理网络的种种限制越来越不适应虚拟化的要求，由此提出了 VXLAN、NVGRE 等网络 Overlay 方案，在这一方案下，物理网络中的东西向流量类型也逐渐由二层向三层转变，通过增加封装，将网络拓扑由物理二层变为逻辑二层，同时提供逻辑二层的划分管理，更好地满足了多用户的需求。VXLAN、NVGRE 等 Overlay 技术都是通过将 MAC 封装在 IP 之上，实现对物理网络的屏蔽，解决了物理网络 VLAN 数量限制、接入交换机 MAC 表项有限等问题。通过提供统一的逻辑网络管理工具，方便地实现了虚拟机 HA 迁移时网络策略跟随的问题。

随着虚拟化技术的进步，每台物理服务器的虚拟机数量由 8 台提升至 16 台、32 台甚至更多，这使得低延迟的服务器间通信和更高的双向带宽需求变得更加迫切。然而传统的网络核心、汇聚和接入的三层结构，服务器虚拟化后还有一个虚拟交换机层，而随着刀片服务器的广泛应用，刀片式交换机也给网络添加了一层交换。如此之多的网络层次，使得数据中心计算节点间通信延时大幅增加，这就需要网络化架构向扁平化方向发展，最终的目标是在任意两点之间尽量减少网络架构的数目。网络扁平化后，减少了中间层次，对核心设备交换能力要求降低，对于数据中心而言，后续扩容只需要以标准的机柜为单位增加即可。

数据中心的这些变化，对网络提出了更高的要求。Fabric 物理网络架构成为解决上述难题的一个重要手段。Fabric 网络架构就是通过骨干网络节点间分层互联或全互联的方式，提供所有接入网络各类节点间的无差异互访。在 Fabric 架构下，所有节点可以全互联，也可以分层互联。分层结构下，节点类型分为骨干节点和叶子节点，骨干节点与叶子节点间全连接，骨干节点仅用作转发，叶子节点作为二三层的边界。在这种架构下，网络全互联形成的大量等价路径既保证了链路的冗余可靠，又提高了整个 Fabric 网络的吞吐量，扁平的网络结构保证了任意节点间较高的连接速率，同时对任意类型流量均拥有极低的时延。

未来 SDN¹⁵提供了可行的解决方案。它通过集中的控制器来实现对整网设备的监控和管理，利用软件的灵活、动态可扩展，提供丰富的管理控制策略，通过开放相关的 API，可以集成第三方 APP，实现更多的个性化的网络控制。SDN 网络是一种全新的网络。在这样的网络中，控制器就是大脑，它掌控全局，学习整网的拓扑，管理网络中的各个节点。网络中的其他节点，只需要向“大脑”上报网络变化，并按照“大脑”的指挥，完成自己的工作即可。SDN 的发展存在不确定性，传统的南向接口 OpenFlow 会越来越复杂，现有网络设备无法简单升级支持，同时满足大流表的需求将大幅提高芯片成本；控制器发展不统一，各个厂家都在争夺自己在未来 SDN 网络的话语权，标准化进行缓慢，各种私有的协议加入导致未来网络成为少数玩家的地盘。但是，无论具体实现形式如何，SDK 这种集中管控、灵活动态的网络部署必将成为未来的发展趋势。

¹⁵ 软件定义网络 (Software Defined Network, SDN)，是 Emulex 网络一种新型网络创新架构，是网络虚拟化的一种实现方式，其核心技术 OpenFlow 通过将网络设备控制面与数据面分离开来，从而实现了网络流量的灵活控制，使网络作为管道变得更加智能。

2.2 新兴技术

1944年的时候,最好的人类计算机每隔几秒钟仅可以实现一次加法或乘法。因此,一般的解决方案是将问题分解为较小的任务,这些小任务可以由不同的人同时执行,每次以相同的计算复杂度运行乘法计算。

70年后,计算机架构师面临类似的挑战,并且采用了类似的解决方案。虽然单个计算设备计算速度很快,但是物理约束对其速度仍然有限制。因此,如今的计算趋势是普适并行计算。单处理器包含流水线、并行指令、预测执行和多线程。本质上,从台式计算机到强大的超级计算机,每个计算机系统都包含多处理器。

通常来讲,分布式计算和集中式计算相反。并行计算领域与分布式计算在很大程度上有交叠,云计算与分布式计算、集中式计算、并行计算都有一部分的交集。

集中式计算:这种计算模式是将所有计算资源集中在一个物理系统之内。所有资源(处理器、内存、存储器)是全部共享的,并且紧耦合在一个集成式的操作系统中。许多数据中心和超级计算机都是集中式系统,但它们都被用于并行计算、分布式计算和云计算应用中。

并行计算:在并行计算中,所有处理器或是紧耦合于中心共享内存或是松耦合于分布式内存。一些学者称之为并行处理。处理器间通信通过共享内存或通过消息传递完成。通常称有并行计算能力的计算系统为并行计算机。运行在并行计算机上的程序称为并行程序。编写并行程序的过程称为并行编程。

分布式计算:这是一个计算机科学和工程中研究分布式系统的领域。一个分布式系统由众多自治的计算机组成,各自拥有其私有内存,通过计算机网络通信。分布式系统中的信息交换通过消息传递的方式完成。运行在分布式系统上的程序称为分布式程序。编写分布式程序的过程称为分布式编程。

云计算:一个互联网云的资源可以是集中式的也可以是分布式的。云采用分布式计算或并行计算,或两者兼有。云可以在集中的或分布式的大规模数据中心之上,由物理的或虚拟的计算资源构建。

普适计算:指在任何地点和时间通过有线或者无线网络使用普遍的设备进行计算。物联网是一个日常生活对象(包括计算机、传感器、人等)网络化的连接。物联网通过互联网云实现任何对象在任何地点和时间的普适计算。

关于云计算技术、大数据相关技术、分布式技术、虚拟化技术以及物联网技术,不再一一介绍,相关内容可参阅本书附赠资源(可到“博文视点”网站下载)。

3 chapter

第 3 章 Java API 调用优化建议

某一年，临近黄河岸畔有一片村庄，为了防止黄患，农民们筑起了巍峨的长堤。一天有个老农偶然发现蚂蚁窝一下子猛增了许多。老农心想这些蚂蚁窝究竟会不会影响长堤的安全呢？他要回村去报告，路上遇见了他的儿子。老农的儿子听了不以为然说：偌坚固的长堤，还害怕几只小小蚂蚁吗？拉老农一起下田了，当天晚上风雨交加，黄河里的水猛涨起来，咆哮的河水从蚂蚁窝渗透出来，继而喷射，终于堤决人淹。这个故事对应的典故是《韩非子·喻老》一文中：“千丈之堤，以蝼蚁之穴溃；百尺之室，以突隙之炽焚。”现代成语对应的是“千里之堤，溃于蚁穴千里之行，始于足下”。

在深入了解 Java 并行计算、JVM 性能优化这些高级优化策略之前，我们首先要会写代码，如果连基本的 Java API 都不会调用，那何来性能优化一说，又何来进一步的针对云计算、虚拟机这些新兴技术的 Java 程序性能优化策略。本章不会从 API 的基础讲起，本章力求让读者能够明白底层的实现、最优的代码编写方式。确实最近推出的 JVM 越来越 Smart，特别是 JIT 的产生，会在编译的时候帮助我们整理性能不好的代码，但是，我个人觉得只有自己了解才算真正懂。而且，这样也可以节省编译时间。

本章主要介绍和解决以下问题，这些也是性能优化深入学习之前的基础知识：

- 如何对数据结构相关代码进行优化。
- 如何对字符串相关操作代码进行优化。
- 如何对引用类型相关代码进行优化。
- 如何采用其他一些技巧。
- 如何从实际范例里学习到优化方法。
- 为后续章节做好编码层面知识准备。

3.1 面向对象及基础类型

3.1.1 采用 Clone()方式创建对象

Java 语言里面的所有类都默认继承自 `java.lang.Object` 类，在 `java.lang.Object` 类里面有一个 `clone()` 方法，JDK API 的说明文档里面解释了这个方法会返回 `Object` 对象的一个拷贝。我们需要说明两点：一是拷贝对象返回的是一个新对象，而不是一个对象的引用地址；二是拷贝对象与用 `new` 关键字操作符返回的新对象的区别是，这个拷贝已经包含了一些原来对象的信息，而不是对象的初始信息，即每次拷贝动作不是一个针对全新对象的创建。

当我们使用 `new` 关键字创建类的一个实例时，构造函数中的所有构造函数都会被自动调用。但如果一个对象实现了 `Cloneable` 接口，那么我们可以通过调用它的 `clone()` 方法，注意，`clone()` 方法不会调用任何构造函数。

代码清单 3-1 所示是工厂模式的一个典型实现，工厂模式是采用工厂方法代替 `new` 操作的一种模式，所以工厂模式的作用就相当于创建实例对象的 `new` 操作符。

代码清单 3-1 创建新对象

```
public static Credit getNewCredit()
{
    return new Credit(); // 创建一个新的 Credit 对象
}
```

如果我们采用 `clone()` 方法的方式创建对象，那么原有的信息可以被保留，因此创建速度会加快。如清单 3-2 所示，改进后的代码使用了 `clone()` 方法。

代码清单 3-2 使用了 clone()方法

```
private static Credit BaseCredit = new Credit();
public static Credit getNewCredit()
{
    return (Credit)BaseCredit.clone();
}
```

3.1.2 避免对 boolean 判断

Java 里的 `boolean` 数据类型被定义为存储 8 位 (1 个字节) 的数值形式，但只能是 `true` 或是 `false`。

有些时候我们出于写代码的习惯，经常容易导致习惯性思维，这里指的习惯性思维是想要对生成的数据进行判别，这样感觉可以在该变量进入业务逻辑之前有一层检查、判定。对于大多数的数据类型来说，这是正确的做法，但是对于 `boolean` 变量，我们应该尽量避免不必要的等于判定。如果尝试去掉 `boolean` 与 `true` 的比较判断代码，大体上来说，我们会有两个好处。

- 代码执行得更快 (生成的字节码少了 5 个字节)；
- 代码整体显得更加干净。

例如代码清单 3-3 和 3-4 所示，我们针对这个判定进行了代码解释，这两个类只有一个差距，即是否调用了等号表达式进行了一致性判定，如代码 `string.endsWith("a") == true`。

代码清单 3-3 boolean 示例 1

```
boolean method (string string) {  
    return string.endswith ("a") == true;//判断是否以 a 结尾  
}
```

代码清单 3-4 boolean 示例 2

```
boolean method (string string) {  
    return string.endswith ("a");  
}
```

3.1.3 多用条件操作符

我们在编写代码的过程中很喜欢使用 if-else 用于判定，这种思维来源于 C 语言学习的经历。大多数中国学生都是从谭老师的 C 语言书籍¹了解计算机领域知识的，我们在高级语言程序设计过程中，如果有可能，尽量使用条件操作符"if (cond) return; else return;"这样的顺序判断结构，主要原因还是因为条件操作符更加简捷，代码看起来会少一点。其实 JVM 会帮助我们优化代码，但是个人感觉能省就省吧，代码过多让人看着不爽。代码清单 3-5 和 3-6 所示是示例代码，对比了两者的区别。

代码清单 3-5 if 示例 1

```
//采用 if-else 的方式  
public int method(boolean isdone){  
    if (isdone) {  
        return 0;  
    } else {  
        return 1;  
    }  
}
```

代码清单 3-6 if 示例

```
public int method(boolean isdone) {  
    return (isdone ? 0 : 1);  
}
```

上面两个例子，我们可以看到有一定差距，代码行数缩短了 50%。其实现代 JVM 已经在编译时做了类似的处理，但是从代码整洁度考虑，我觉得还是推荐多采用代码清单 3-6 的方式实现。

3.1.4 静态方法替代实例方法

在 Java 中，使用 static 关键字描述的方法是静态方法。与静态方法相比，实例方法的调用需要消耗更多的系统资源，这是因为实例方法需要维护一张类似虚拟函数导向表的结构，这样可以方便地实现对多态的支持。对于一些常用的工具类方法，我们没有必要对其进行重载，那么我们可以尝试将它们声明为 static，即静态方法，这样有利于加速方法的调用。

¹ 即谭浩强教授，他编著的《C 程序设计》发行了 1100 万册。

如代码清单 3-7 所示，我们分别定义了两个方法，一个是静态方法，一个是实例方法，然后在 main 函数进程里分别调用 10 亿次两个方法，计算两个方法的调用总计时间。

代码清单 3-7 静态方法示例

```
public static void staticMethod(){
}
//实例方法
public void instanceMethod(){

}

@Test
public static void main(String[] args){
    long start = System.currentTimeMillis();
    //循环 10 亿次，创建静态方法
    for(int i=0;i<1000000000;i++){
        staticVSInstance.staticMethod();
    }

    System.out.println(System.currentTimeMillis() - start);

    start = System.currentTimeMillis();
    staticVSInstance sil = new staticVSInstance();
    //循环 10 亿次，创建实例方法
    for(int j=0;j<1000000000;j++){
        sil.instanceMethod();
    }
    System.out.println(System.currentTimeMillis() - start);
}
```

清单 3-7 代码中申明了一个静态方法 `staticMethod()` 和一个实例方法 `instanceMethod()`，运行程序，统计了两个方法调用若干次后的耗时，程序输出如下，单位是毫秒，方法内部没有实现任何代码。请读者注意，由于机器差别，所以运行的结果可能也会有所不同。

代码清单 3-8 程序运行输出

733 764

总的来说，静态方法和实例方法的区别主要体现在以下两个方面。

- 在外部调用静态方法时，可以使用“类名.方法名”的方式，也可以使用“对象名.方法名”的方式。而实例方法只有后面这种方式。也就是说，调用静态方法可以无须创建对象。
- 静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法），而不允许访问实例成员变量和实例方法；实例方法则无此限制。

从上面的例子我们可以这么总结，如果你没有必要去访问对象的外部，那么就让你的方法成为静态方法。静态方法会被更快地调用，因为它不需要一个虚拟函数导向表，该表用来告诉你如何区分方法的性质，调用这个方法不会改变对象的状态。

3.1.5 有条件地使用 final 关键字

在 Java 中, final 关键字可以被用来修饰类、方法和变量(包括成员变量和局部变量)。我们在使用匿名内部类的时候可能会经常用到 final 关键字,例如 Java 中的 String 类就是一个 final 类。

如代码清单 3-9 所示,由于 final 关键字会告诉编译器,这个方法不会被重载,所以我们可以让访问实例内变量的 getter/setter 方法变成“final”。

代码清单 3-9 非 final 类

```
public void setsize (int size) {
    _size = size;
}
private int _size;
```

代码清单 3-10 final 类

```
//告诉编译器该方法不会被重载
final public void setsize (int size) {
    _size = size;
}
private int _size;
```

总的来说,使用 final 方法的原因有两个²。第 1 个原因是把方法锁定,以防任何继承类修改它的含义。第 2 个原因是提高效率。在早期的 Java 实现版本中,会将 final 方法转为内嵌调用。但是如果方法过于庞大,可能看不到内嵌调用带来的任何性能提升。JDK6 以后的 Java 版本已经不再需要使用 final 方法进行这些优化了。

3.1.6 避免不需要的 instanceof 操作

instanceof 关键字是 Java 的一个二元操作符,和 ==、>、< 是属于同一类表达式。由于 instanceof 是由字母组成的,所以它也是 Java 的保留关键字。instanceof 的作用是测试它左边的对象是否是它右边的类的实例,返回 boolean 类型的数据,即如果左边的对象的静态类型等于右边的,我们使用的 instanceof 表达式的返回值会返回 true。

代码清单 3-11 instanceof 示例

```
void method (dog dog, faClass faClass) {
    dog d = dog;
    if (d instanceof faClass) // 这里永远都返回 true.
        system.out.println("dog is a faClass ");
    faClass faClass = faClass;
    if (faClass instanceof object) // always true.
        system.out.println("uiso is an object");
}
```

上述代码里面对 dog 类型的变量都做了判定,由于已经确定类继承自基类,所以我们可以删

² 这段话摘自《Java 编程思想》第四版第 143 页。

除不需要的 instanceof 操作。当然，这样的操作修改还是需要基于实际的业务逻辑，有些时候为了保证数据准确性、安全性，还是需要层层检查的。如代码清单 3-12 所示，代码可以被精简成这样。

代码清单 3-12 instanceof 示例

```
void method () {
    dog d;
    system.out.println ("dog is an faclass");
    system.out.println ("uiso is an faclass");
}
```

另外，绝大多数情况下都不推荐使用 instanceof 方法，还是好好利用多态特性吧，这是面向对象的基本功能。

3.1.7 避免子类中存在父类转换

我们知道在 Java 语言里所有的类都是直接或者间接继承自 Object 类。我们可以说，Object 类是所有 Java 类的祖先，因此每个类都使用 Object 作为超类，所有对象（包括数组）都实现这个类的方法。在不明确是否提供了超类的情况下，Java 会自动把 Object 作为被定义类的超类。

我们可以使用类型为 Object 的变量指向任意类型的对象。同样，所有的子类也都隐含的“等于”其父类。那么，程序代码中就没有必要再把子类对象转换为它的父类了。

代码清单 3-13 避免父类转换示例

```
class oriClass {
    string _id = "unc";
}
class dog extends oriClass{
    void method () {
        dog dog = new dog();
        oriClass animal = (oriClass)dog; //已经确定继承自 oriClass 类了，因此没有必要再转
对象类型
        object o = (object)dog;
    }
}
```

代码清单 3-14 避免父类转换示例

```
class dog extends oriClass {
    //去掉了转换父类操作
    void method () {
        dog dog = new dog();
        unc animal = dog;
        object o = dog;
    }
}
```


3.1.8 建议多使用局部变量

调用方法时传递的参数以及在调用中创建的临时变量都被保存在栈（Stack）里面，因此读写速度较快。其他变量，例如静态变量、实例变量，它们都在堆（heap）中被创建，也被保留在那里，所以读写相对于保存在栈里面的数据来说，它的速度较慢。

Java 类的成员变量有两种，一种是被 static 关键字修饰的变量，叫类变量或者静态变量，另一种没有 static 修饰，称为实例变量。在语法定义上的区别，静态变量前要加 static 关键字，而实例变量前则不加。

静态变量（类变量）被所有对象共有，如果其中一个对象将它的值改变，那么其他对象得到的就是改变后的结果。实例变量属于对象私有，如果某一个对象将其值改变，也不会影响到其他对象。此外，静态变量和实例变量都属全局变量。

程序运行过程当中，实例变量属于某个对象的属性，必须创建实例对象，其中的实例变量才会被分配空间，才能使用这个实例变量。静态变量不属于某个实例对象，而是属于类，所以也称为类变量，只要程序加载了类的字节码，不用创建任何实例对象，静态变量就会被分配空间，静态变量就可以被使用了。总之，实例变量必须创建对象后才可以通过这个对象来使用，静态变量则可以直接使用类名来引用。

代码 3-15 演示了分别创建 100 万次实例变量和静态变量所消耗的时间，从测试结果来看，使用局部变量和静态变量的操作时间对比较为明显。

代码清单 3-15 局部变量和静态变量之间的对比测试

```
public class variableCompare {
    public static int b = 0;
    public static void main(String[] args){
        int a = 0;
        long starttime = System.currentTimeMillis();
        for(int i=0;i<1000000;i++){
            a++;//在函数体内定义局部变量
        }
        System.out.println(System.currentTimeMillis() - starttime);
        starttime = System.currentTimeMillis();
        for(int i=0;i<1000000;i++){
            b++;//在函数体内定义局部变量
        }
        System.out.println(System.currentTimeMillis() - starttime);
    }
}
```

以上两段代码的运行时间分别为 0 和 15，单位是毫秒。由此结果可见，局部变量的访问速度远远高于类的成员变量。

3.1.9 运算效率最高的方式——位运算

在 Java 语言中的所有运算中，位运算是最为高效的。位运算表达式由操作数和位运算符组成，实现对整数类型的二进制数进行位运算。位运算符可以分为逻辑运算符（包括~、&、|和^）及移

位运算符（包括>>、<<和>>>）。因此，可以尝试使用位运算方式代替部分算术运算，来提高系统的运行速度。最典型示例的就是对于整数的乘除运算优化。

代码清单 3-16 实现了位运算与算术运算的对比，两个运算方式输出的结果是一样的，但是耗时差距达到了 8 倍。

代码清单 3-16 位运算与算术运算对比试验

```
public class yunsuanClass {
    public static void main(String args[]){
        long start = System.currentTimeMillis();
        long a=1000;
        //执行 1000 万次算术运算
        for(int i=0;i<10000000;i++){
            a*=2;
            a/=2;
        }
        System.out.println(a);
        System.out.println(System.currentTimeMillis() - start);
        start = System.currentTimeMillis();
        //执行 1000 万次位运算
        for(int i=0;i<10000000;i++){
            a<<=1;
            a>>=1;
        }
        System.out.println(a);
        System.out.println(System.currentTimeMillis() - start);
    }
}
```

两段代码执行了完全相同的功能，在每次循环中，整数 1000 乘以 2，然后除以 2。第 1 个循环耗时 546，第 2 个循环耗时 63，单位是毫秒（ms）。

我们的程序内部进行位运算时，需要注意以下几点。

- >>>和>>的区别是：在执行运算时，>>>运算符的操作数高位补 0，而>>运算符的操作数高位移入原来高位的值。
- 右移一位相当于除以 2，左移一位（在不溢出的情况下）相当于乘以 2；移位运算速度高于乘除运算。
- 若进行位逻辑运算的两个操作数的数据长度不相同，则返回值应该是数据长度较长的数据类型。
- 按位异或可以不使用临时变量完成两个值的交换，也可以使某个整型数的特定位的值翻转。
- 按位与运算可以用来屏蔽特定的位，也可以用来取某个数型数中某些特定的位。
- 按位或运算可以用来对某个整型数的特定位的值置 1。

个人建议，由于移位操作需要一定的底层编程技术能力，所以对于刚开始接触程序设计的读

者来说,除非是在一个非常大的循环内,性能因素至关重要,而且你很清楚你自己在做什么,才建议使用这种方法,否则提高性能所带来的程序易读性的降低就不划算了。

3.1.10 用一维数组代替二维数组

JDK 很多类库是采用数组方式实现的数据存储,比如 ArrayList、Vector 等,数组的优点是随机访问性能非常好。

一维数组和二维数组的访问速度不一样,二维数组的访问速度要优于一维数组,但是,二维数组比一维数组占用更多的内存空间,大概是 10 倍左右。在性能敏感的系统中使用二维数组,如果内存不足,尽量将二维数组转化为一维数组再进行处理,以节省内存空间。

如代码清单 3-17 所示,我们演示了一维数组和二维数组比较的示例程序。

代码清单 3-17 一维数组和二维数组对比

```
public class arrayTest {
    public static void main(String[] args){
        long start = System.currentTimeMillis();
        int[] arraySingle = new int[1000000];
        int chk = 0;
        //构建 1 亿个数组元素,并赋值
        for(int i=0;i<100;i++){
            for(int j=0;j<arraySingle.length;j++){
                arraySingle[j] = j;
            }
        }
        //遍历 1 亿个数组元素,并赋值给局部变量
        for(int i=0;i<100;i++){
            for(int j=0;j<arraySingle.length;j++){
                chk = arraySingle[j];
            }
        }
        System.out.println(System.currentTimeMillis() - start);

        start = System.currentTimeMillis();
        int[][] arrayDouble = new int[1000][1000];
        chk = 0;
        //构建对应于 1 亿个一维数组的二维数组
        for(int i=0;i<100;i++){
            for(int j=0;j<arrayDouble.length;j++){
                for(int k=0;k<arrayDouble[0].length;k++){
                    arrayDouble[i][j]=j;
                }
            }
        }
        //遍历这些二维数组
        for(int i=0;i<100;i++){
            for(int j=0;j<arrayDouble.length;j++){
```

```

        for(int k=0;k<arrayDouble[0].length;k++){
            chk = arrayDouble[i][j];
        }
    }
}
System.out.println(System.currentTimeMillis() - start);

start = System.currentTimeMillis();
arraySingle = new int[1000000];
int arraySingleSize = arraySingle.length;
chk = 0;
//遍历一维数组
for(int i=0;i<100;i++){
    for(int j=0;j<arraySingleSize;j++){
        arraySingle[j] = j;
    }
}
for(int i=0;i<100;i++){
    for(int j=0;j<arraySingleSize;j++){
        chk = arraySingle[j];
    }
}
System.out.println(System.currentTimeMillis() - start);

start = System.currentTimeMillis();
arrayDouble = new int[1000][1000];
int arrayDoubleSize = arrayDouble.length;
int firstSize = arrayDouble[0].length;
chk = 0;
//遍历二维数组
for(int i=0;i<100;i++){
    for(int j=0;j<arrayDoubleSize;j++){
        for(int k=0;k<firstSize;k++){
            arrayDouble[i][j]=j;
        }
    }
}
for(int i=0;i<100;i++){
    for(int j=0;j<arrayDoubleSize;j++){
        for(int k=0;k<firstSize;k++){
            chk = arrayDouble[i][j];
        }
    }
}
System.out.println(System.currentTimeMillis() - start);
}
}

```


第一段代码操作的是一维数组的赋值、取值过程，第二段代码操作的是二维数组的赋值、取值过程，第三段代码是一维数组遍历、赋值过程，第四段代码是二维数组的遍历、赋值过程。输出时间分别是 374、312、297、266 毫秒。从 3-17 所示代码的运行结果来看，二维数组的速度有一定优势，但是请注意，这是 JVM 牺牲了内存空间换取的性能，请读者自己做出选择。

3.1.11 布尔运算代替位运算

虽然位运算的速度远远高于算术运算，但是在条件判断时，使用位运算替代布尔运算是非常错误的选择。在条件判断时，Java 会对布尔运算做相当充分的优化。假设有表达式 a、b、c 进行布尔运算 “a&&b&&c”，根据逻辑与的特点，只要在整个布尔表达式中有一项返回 false，整个表达式就返回 false，因此，当表达式 a 为 false 时，该表达式将立即返回 false，而不会再去计算表达式 b 和 c。若此时，表达式 a、b、c 需要消耗大量的系统资源，这种处理方式可以节省这些计算资源。同理，当计算表达式 “a||b||c” 时，只要 a、b 或 c，3 个表达式其中任意一个计算结果为 true 时，整体表达式立即返回 true，而不去计算剩余表达式。简单地说，在布尔表达式的计算中，只要表达式的值可以确定，就会立即返回，而跳过剩余子表达式的计算。如果使用位运算(按位与、按位或)代替逻辑与和逻辑或，虽然位运算本身没有性能问题，但是位运算总是要将所有的子表达式全部计算完成后，再给出最终结果。因此，从这个角度看，使用位运算替代布尔运算会使系统进行很多无效计算。代码清单 3-18 演示了位运算与布尔运算的对比实验。

代码清单 3-18 位运算与布尔运算的比较

```
public class OperationCompare {
    public static void booleanOperate(){
        long start = System.currentTimeMillis();
        boolean a = false;
        boolean b = true;
        int c = 0;
        //下面循环开始进行位运算，表达式里面的所有计算因子都会被用来计算
        for(int i=0;i<1000000;i++){
            if(a&b&"Test_123".contains("123")){
                c = 1;
            }
        }
        System.out.println(System.currentTimeMillis() - start);
    }

    public static void bitOperate(){
        long start = System.currentTimeMillis();
        boolean a = false;
        boolean b = true;
        int c = 0;
        //下面循环开始进行布尔运算，只计算表达式 a 即可满足条件
        for(int i=0;i<1000000;i++){
            if(a&&b&&"Test_123".contains("123")){
                c = 1;
            }
        }
    }
}
```

```

    }
    System.out.println(System.currentTimeMillis() - start);
}

public static void main(String[] args){
    OperationCompare.booleanOperate();
    OperationCompare.bitOperate();
}
}

```

上面的示例代码运行结果显示布尔计算大大优于位运算（位运算用了 63 毫秒，布尔运算几乎没有耗时），但是，这个结果不能说明位运算比逻辑运算慢，因为在所有的逻辑与运算中，都省略了表达式 “`"Test_123".contains("123")`” 的计算，而所有的位运算都没能省略这部分系统开销。

3.1.12 提取表达式优化

在大部分情况下，由于计算机运算单元(CPU)的不断发展，我们现在用的 CPU 大多是多核的，有些可能还会附带 GPU，这样我们可以把图像计算、数据挖掘算法这类需要消耗大量计算资源的程序放到 GPU 上运行，这是题外话了，不多展开。这样，我们知道在程序高速运行过程当中，少量的重复代码并不会对性能构成太大的威胁，但是如果你希望将系统性能发挥到极致，则还是有很多地方可以优化的。比如代码清单 3-19 所示的代码，我们通过采用局部变量的方式，避免了重复的计算，虽然计算量相对于 CPU 来说很微小，但是总是还是可以节省一点时间的。

代码清单 3-19 提取表达式实验

```

public class duplicatedCode {
    public static void beforeTuning(){
        long start = System.currentTimeMillis();
        double a1 = Math.random();
        double a2 = Math.random();
        double a3 = Math.random();
        double a4 = Math.random();
        double b1,b2;
        //开始循环运算
        for(int i=0;i<10000000;i++){
            b1 = a1*a2*a4/3*4*a3*a4;
            b2 = a1*a2*a3/3*4*a3*a4;
        }
        System.out.println(System.currentTimeMillis() - start);
    }

    public static void afterTuning(){
        long start = System.currentTimeMillis();
        double a1 = Math.random();
        double a2 = Math.random();
        double a3 = Math.random();
        double a4 = Math.random();
        double combine,b1,b2;
    }
}

```



```
//计算公式被移到了外面
for(int i=0;i<10000000;i++){
    combine = a1*a2/3*4*a3*a4;
    b1 = combine*a4;
    b2 = combine*a3;
}
System.out.println(System.currentTimeMillis() - start);
}

public static void main(String[] args){
    duplicatedCode.beforeTuning();
    duplicatedCode.afterTuning();
}
}
```

两段代码的差别是提取了重复的攻势，使得这个公式的每次循环计算只执行一次。分别耗时 202ms 和 110ms，可见，提取复杂的重复操作是相当具有意义的。这个例子告诉我们，在循环体内，如果能够提取到循环体外的计算公式，最好提取出来，尽可能让程序少做重复的计算。

3.1.13 不要总是使用取反操作符(!)

取反操作符(!)表示异或操作，使用起来很方便，但是也要注意的，它降低了程序的可读性，所以建议不要经常使用。

代码清单 3-20 取反操作符示例

```
boolean method (boolean a, boolean b) {
    if (!a)
        return !a;
    else
        return !b;
}
```

3.1.14 不要重复初始化变量

默认情况下，调用类的构造函数时，Java 会把变量初始化为一个确定的值，例如，所有的对象被设置成 Null，整数变量设置成 0，float 和 double 变量设置成 0.0，逻辑值设置成 false。当一个类从另一个类派生时，这一点尤其应该注意，因为用 new 关键字创建一个对象时，构造函数链中的所有构造函数都会被自动调用。

这里需要注意，当我们给成员变量设置初始值，又需要调用其他方法的时候，最好放在一个方法里面。比如 initXXX() 中，因为直接调用某方法赋值可能会因为类尚未初始化而抛出指针异常。

此外，如果不初始化变量，那么当我们直接调用变量的时候，系统会给对象或变量随机赋一个值，这样容易产生不必要的错误。

3.1.15 变量初始化过程思考

由于智能化的 GUI 工具，例如 Eclipse、IntelliIDEA 这样的工具存在，所以程序员很少会去主

动思考 Java 程序对于成员变量的声明及初始化顺序，一般都是按照自己的习惯方式进行编码，如果出错了 GUI 工具也会自动提醒，点一下左侧的修复按钮就会自动修复代码。

我们思考一个问题，为什么抽象类不能用 `final` 关键字声明？如果读者看过闫宏博士的《Java 与模式》书，那么可能已经知道了答案，因为一个类一旦被修饰成了 `final`，那么意味着这个类是不能被继承的，并且 Java 中定义抽象类不能被实例化。如果一个抽象类可以是 `final` 类型的，那么这个类就不能被继承，也不能被实例化，那么它也就没有存在的意义了。即从语言的角度来讲，一个类既然是抽象类，那么它就是为了被其他类继承，所以给它标识为 `final` 是没有意义的。

我们来看一系列的示例代码，通过这些代码可以帮助读者理解成员变量的初始化过程。

如代码清单 3-21 所示，我们首先定义局部变量 `variableA` 的值为 1，然后申明变量，由于 `main` 主函数里面会实例化类 `varClass`，所以 `variableA` 会被赋值为 1。

代码清单 3-21 普通局部变量

```
public class varClass {
    {
        variableA = 1;
    }

    private int variableA;

    public static void main(String[] args){
        3-24Class test1 = new 3-24Class ();
        System.out.println(test1.variableA);
    }
}
```

程序运行输出为 1，如果我们想要打印出 `variableA` 的值，我们必须等到成员变量被申明后才能这么做，代码清单 3-22 所示的代码，GUI 工具会提醒我们，“cannot reference a field before it is defined”，也就是无法在申明之前调用它的引用地址。

代码清单 3-22 普通局部变量尝试输出

```
public class errorClass
{
    variableA = 1;
    System.out.println(variableA); //这一行会抛出错误，必须注释后才能在 GUI 工具里面运行
    这个类

    private final int variableA;

    public static void main(String[] args){
        errorClass test1 = new errorClass();
        System.out.println(test1.variableA);
    }
}
```


如果把 `variableA` 申明为 `final` 呢？如代码清单 3-23 所示。

代码清单 3-23 `final` 局部变量

```
public class finalClass {
    {
        variableA = 1;
    }

    private final int variableA;

    public static void main(String[] args){
        finalClass test1 = new finalClass();
        System.out.println(test1.variableA);
    }
}
```

输出依然是 1，但是如果我们尝试对 `final` 关键字申明的 `variableA` 变量赋值，那么又会产生错误 “The final field Class1.variableA cannot be assigned”，即既然是不可变的变量，又怎么允许在申明之后的代码块里面继续变更变量值呢。

如果我们对 3-22 代码稍作修改，定义 `private int variable=2`，程序的输出会变成 2，这是因为代码的分析、执行是从上至下的，实质上我们可以看出，JVM 将变量 `variable` 的申明和赋值分为了两个步骤，即先执行申明操作，然后顺序地执行第 1 步赋值操作，即赋值为 1，然后执行第 2 步赋值操作，赋值为 2，这样最终类实例会打印出第 2 个值，即 2。

如果我们依然想打印出 1 呢？我们可以通过申明变量为静态变量的方式来达到这样的目的，如代码清单 3-24 所示。

代码清单 3-24 `static` 变量

```
public class staticClass {
    //static
    {
        variableA = 1;
    }

    private static int variableA = 2;

    public static void main(String[] args){
        staticClass test1 = new staticClass();
        System.out.println(test1.variableA);
    }
}
```

如果希望输出为 2，那么可以释放 3-23 代码中注释的针对代码块的 `static` 关键字申明，这样可以确保程序先执行静态代码块，也就确保了赋值 2 的定义被后续执行。

这个小节讲述的是成员变量、代码块的申明过程、执行顺序，我们在很多场景下可能不会

去思考这类问题，但是作者觉得还是有必要思考的，这样可以帮助程序员理清思路，也许，下一个语言是由你创造的呢？

3.1.16 对象的创建、访问过程

■ 对象的创建

创建一个对象通常是需要 `new` 关键字，当虚拟机遇到一条 `new` 指令时，首先检查这个指令的参数是否在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已被加载、解析和初始化过。如果那么执行相应的类加载过程。

类加载检查通过后，虚拟机将为新生对象分配内存。为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。分配的方式有两种，一种叫指针碰撞，假设 Java 堆中内存是绝对规整的，用过的和空闲的内存各在一边，中间放着一个指针作为分界点的指示器，分配内存就是把那个指针向空闲空间的那边挪动一段与对象大小相等的距离。另一种叫空闲列表，如果 Java 堆中的内存不是规整的，虚拟机就需要维护一个列表，记录哪个内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录。采用哪种分配方式是由 Java 堆是否规整决定的，而 Java 堆是否规整是由所采用的垃圾收集器是否带有压缩整理功能决定的。另外一个需要考虑的问题就是对象创建时的线程安全问题，有两种解决方案：一是对分配内存空间的动作进行同步处理；另一种是把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在 Java 堆中预先分配一小块内存(TLAB)，哪个线程要分配内存就在哪个线程的 TLAB 上分配，只有 TLAB 用完并分配新的 TLAB 时才需要同步锁定。

内存分配完成后，虚拟机需要将分配到的内存空间初始化为零值。这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就可以直接使用。接下来虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息等，这些信息存放在对象的对象头中。

上面的工作都完成以后，从虚拟机的角度来看一个新的对象已经产生了。但是从 Java 程序的角度，还需要执行 `init` 方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

■ 对象的内存布局

在 HotSpot 虚拟机中，对象在内存中存储的布局可分为三个部分，即对象头、实例数据和对齐填充。

对象头包括两个部分：第一部分用于存储对象自身的运行时数据，如哈希码、GC 分代年龄、线程所持有的锁等。官方称之为“Mark Word”。第二个部分为是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

实例数据是对象真正存储的有效信息，也是程序代码中所定义的各种类型的字段内容。

对齐填充并不是必然存在的，仅仅起着占位符的作用。Hotspot VM 要求对象起始地址必须是 8 字节的整数倍，对象头部分正好是 8 字节的倍数，所以当实例数据部分没有对齐时，需要通过对齐填充来对齐。

■ 对象的访问定位

Java 程序通过栈上的 reference 数据来操作堆上的具体对象。主要的访问方式有使用句柄和直接指针两种：

- 句柄：Java 堆将会划出一块内存来作为句柄池，引用中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。
- 直接指针：Java 堆对象的布局要考虑如何放置访问类型数据的相关信息，引用中存储的就是对象地址。

两个方式各有优点，使用句柄最大的好处是引用中存储的是稳定的句柄地址，对象被移动时只会改变句柄中实例的地址，引用不需要修改、使用直接指针访问的好处是速度更快，它节省了一次指针定位的时间开销。

3.1.17 在 switch 语句中使用字符串

对于 switch 语句，开发人员并不陌生。大部分编程语言中都有类似的语法结构，用来根据某个表达式的值选择要执行的语句块。对于 switch 语句中的条件表达式类型，不同编程语言所提供的支持是不一样的。对于 Java 语言来说，在 Java 7 之前，switch 语句中的条件表达式的类型只能是与整数类型兼容的类型，包括基本类型 char、byte、short 和 int，与这些基本类型对应的封装类 Character、Byte、Short 和 Integer，还有枚举类型。这样的限制降低了语言的灵活性，使开发人员在需要根据其他类型的表达式来进行条件选择时，不得不增加额外的代码来绕过这个限制。为此，Java 7 放宽了这个限制，额外增加了一种可以在 switch 语句中使用的表达式类型，那就是很常见的字符串，即 String 类型。

Java 7 新特性并没有改变 switch 的语法含义，只是多了一种开发人员可以选择的条件判断的数据类型。但是这个简单的新特性却带来了重大的影响，因为根据字符串进行条件判断在开发中是很常见的。

考虑这样一个应用场景，在程序中需要根据用户的性别来生成合适的称谓。判断条件的类型可以是字符串，不过这在 Java 7 之前的 switch 语句中是行不通的，之前只能添加额外的代码先将字符串转换成整数类型。而在 Java 7 中就可以根据字符串进行条件判断，代码如清单 3-25 所示。

代码清单 3-25 Switch 新特性

```
public class Title {  
    public String generate(String name,String gender){  
        String title = "";  
        switch(gender) {  
            case "男":  
                title = name + "先生";  
                break;  
            case "女":  
                title = name + "女士";  
                break;  
            default:  
                title = name;  
        }  
    }  
}
```

```

    }
    return title;
}
}

```

在 switch 语句中，表达式的值不能是 null，否则会在运行时抛出 NullPointerException。在 case 子句中也不能使用 Null，否则会出现编译错误。

根据 switch 语句的语法要求，其 case 子句的值是不能重复的。这个要求对字符串类型的条件表达式同样适用。不过对于字符串来说，这种重复值得检查还有一个特殊之处，那就是 Java 代码中的字符串可以包含 Unicode 转义字符。重复值的检查是在 Java 编译器对 Java 源代码进行相关的词法转换之后才进行的。这个词法转换过程中包括了对 Unicode 转义字符的处理。也就是说，有些 case 子句的值虽然在源代码中看起来是不同的，但是经词法转换后是一样的，这就会造成编译错误。如下面的代码是无法通过编译的。这是因为其中的 switch 语句中的两个 case 子句所使用的值在经过词法转换之后会变成一样的。

代码清单 3-26 错误的 switch

```

public class Title {
    public String generate(String name,String gender){
        String title = "";
        switch(gender) {
            case "男":
                title = name + "先生";
                break;
            case "\u7537":
                title = name + "女士";
                break;
            default:
                title = name;
        }
        return title;
    }
}

```

代码清单 3-26 所示代码，eclipse 会提示错误：Duplicate case。

通过以上代码的演示，大家应该清楚了新特性的作用。实际上，这个新特性是在编译器这个层次上实现的。而在 Java 虚拟机和字节代码这个层次上，还是只支持在 switch 语句中使用与整数类型兼容的类型。这么做的目的是为了减少这个特性所影响的范围，以降低实现的代价。在编译器层次实现的含义是，虽然开发人员在 Java 源代码的 switch 语句中使用了字符串类型，但是在编译的过程中，编译器会根据源代码的含义来进行转换，将字符串类型转换成与整数类型兼容的格式。不同的 Java 编译器可能采用不同的方式来完成这个转换，并采用不同的优化策略。举个例子，如果 switch 语句中只包含一个 case 子句，那么可以简单地将其转换成一个 if 语句。如果 switch 语句中包含一个 case 子句和一个 default 子句，那么可以将其转换成一个 if-else 语句。而对于复杂的情况，即 switch 语句中包含多个 case 子句的情况，也可以转换成 Java 7 之前的 switch 语句，只不

过使用字符串的哈希值作为 switch 语句的表达式 的值。

为了探究 OpenJDK 中的 Java 编译器使用的是什么样的转换方式,需要一个名为 JAD 的工具。这个工具可以把 Java 的类文件反编译成 Java 的源代码。在对编译生成 Title 类的 class 文件使用了 JAD 之后,所得到的内容如清单 3-27 所示。

代码清单 3-27 编译后的类代码

```
public class Title
{
    public String generate(String name,String gender)
    {
        String title = "";
        String s = gender;
        byte byte0 = -1;
        switch(s.hashCode())
        {
            case 30007:
                if(s.equals("\u7537"))
                    byte0 = 0;
                break;
            case 22899:
                if(s.equals("\u5973"))
                    byte0 = 0;
                break;
        }
        switch(byte0)
        {
            case 0:/'\0'
                title = (new StringBuilder()).append(name).append("\u5148\u751F").toString();
                break;
            case 1:/'\001'
                title = (new StringBuilder()).append(name).append("\u5973\u58EB").toString();
                break;
            default:
                title = name;
                break;
        }
        return title;
    }
}
```

从上面的代码可以看出,原来用在 switch 语句中的字符串被替换成了对应的哈希值,而 case 子句的值也被换成了原来字符串常量的哈希值。经过这样的转换,Java 虚拟机所看到的仍然是与整数类型兼容的类型。在这里值得注意的是,在 case 子句对应的语句块中仍然需要使用 String 的 equals 方法来进行字符串比较。这是因为哈希函数在映射的时候可能存在冲突,多个字符串的哈希值可能是一样的。进行字符串比较是为了保证转换之后的代码逻辑与之前完全一样。

Java 7 引入的这个新特性虽然为开发人员提供了方便，但是比较容易被误用，造成代码的可维护性差的问题。提到这一点就必须要说一下 Java SE 5.0 中引入的枚举类型。switch 语句的一个典型的应用就是在多个枚举值之间进行选择。在 Java SE 5.0 之前，一般的做法是使用一个整数来为这些枚举值编号，比如 0 表示“男”，1 表示“女”。在 switch 语句中使用这个整数编码来进行判断。这种做法的弊端有很多，比如不是类型安全的、没有名称空间、可维护性差和不够直观等。Joshua Bloch 最早在他的“Effective Java”一书中提出了一种类型安全的枚举类型的实现方式。这种方式在 J2SE 5.0 中被引入到标准库，就是现在的 enum 关键字。

Java 语言中的枚举类型的最大优势在于它是一个完整的 Java 类，除了定义其中包含的枚举值之外，还可以包含任意的方法和域，以及实现任意的接口。这使得枚举类型可以很好地与其他 Java 类进行交互。在涉及多个枚举值的情况下，都应该优先使用枚举类型。

在 Java 7 之前，也就是 switch 语句还不支持使用字符串表达式类型时，如果要枚举的值本身都是字符串，使用枚举类型是唯一的选择。而在 Java 7 中，由于 switch 语句增加了对字符串条件表达式的支持，一些开发人员会选择放弃枚举类型而直接在 case 子句中用字符串常量来列出各个枚举值。这种方式虽然简单和直接，但是会带来维护上的麻烦，尤其是这样的 switch 语句在程序的多个地方出现的时候，在程序中多次出现字符串常量总是一个不好的现象，而使用枚举类型就可以避免这种情况。

3.1.18 数值字面量的改进

在编程语言中，字面量（literal）指的是在源代码中直接表示的一个固定的值。绝大部分编程语言都支持在源代码中使用基本类型字面量，包括整数、浮点数、字符串和布尔值等。少数编程语言支持复杂类型的字面量，如数组和对象等。Java 语言只支持基本类型的字面量。Java 7 中对数值类型字面量进行了增强，包括对整数和浮点数字面量的增强。

在 Java 源代码中使用整数字面量的时候，可以指定所使用的进制。在 Java 7 之前，所支持的进制包括十进制、八进制和十六进制。十进制是默认使用的进制。八进制是用在整数字面量之前添加“0”来表示的，而十六进制则是用在整数字面量之前添加“0x”或“0X”来表示的。Java 7 中增加了一种可以在字面量中使用的进制，即二进制。二进制整数字面量是通过在数字前面添加“0b”或“0B”来表示的。

代码清单 3-28 二进制示例 1

```
import static java.lang.System.out;

public class BinaryIntegralLiteral {
    public void display(){
        out.println(0b0011001); //输出 9
        out.println(0B0011110); //输出 14
    }
    public static void main(String[] args){
        BinaryIntegralLiteral b = new BinaryIntegralLiteral();
        b.display();
    }
}
```

这种新的二进制字面量的表示方式使得在源代码中使用二进制数据变得更加简单，不再需要

先手动将数据转换成对应的八/十/十六进制的数值。

如果 Java 源代码中有一个很长的数值字面量，开发人员在阅读这段代码时需要很费力地分辨数字的位数，以知道其所代表的数值大小。在现实生活中，当遇到很长的数字的时候，我们采取的是分段分隔的方式。比如数字 500000，我们通常会写成 500,000，即每三位数字用逗号分隔。利用这种方式就可以很快知道数值的大小。这种做法的理念被加入到了 Java 7 中，不过用的不是逗号，而是下画线 “_”。

在 Java 7 中，数值字面量，不管是整数还是浮点数，都允许在数字之间插入任意多个下画线。这些下画线不会对字面量的数值产生影响，其目的主要是方便阅读。

代码清单 3-29 二进制示例 2

```
import static java.lang.System.out;

public class BinaryIntegralLiteral {
    public void display(){
        out.println(0b001001); //输出 9
        out.println(0B001110); //输出 14
        out.println(1_500_500); //输出 1500500
        out.println(5_6.3_4); //输出 56.34
        out.println(89_3__1); //输出 8931
    }

    public static void main(String[] args){
        BinaryIntegralLiteral b = new BinaryIntegralLiteral();
        b.display();
    }
}
```

虽然下画线在数值字面量中的应用非常灵活，但有些情况是不允许出现的。最基本的原则是下画线只能出现在数字中间，也就是说前后都必须是数字。所以“_100”、“120_”、“ob_101”、“0x_da0”这样的使用方式都是非法的，无法通过编译。这样限制的动机在于降低实现的复杂度。有了这个限制之后，Java 编译器只需要在扫描源代码的时候，将所发现的数字中间的下画线直接删除就可以了。这样就和没有使用下画线的形式是相同的。如果不添加这个限制，那么编译器就需要进行语法分析才能做出判断。比如“_100”可能是一个整数字面量 100，也可能是一个变量名称。这就要求编译器的实现做出更加复杂的改动。

3.1.19 优化变长参数的方法调用

J2SE 5.0 中引入的一个新特性就是允许在方法声明中使用可变长度的参数。一个方法的最后一个形式参数可以被指定为代表任意多个相同类型的参数。在调用的时候，这些参数是以数组的形式来传递的。在方法体中也可以按照数组的方式来引用这些参数。如下代码中可以对多个整数进行求和，可以用类似 sum(1,2,3)这样的形式来调用此方法。

代码清单 3-30 变长参数示例

```
public int sum(int...args){
    int result = 0;
```



```

    for(int value:args){
        result += value;
    }
    return result;
}

```

可变长度的参数在实际开发中可以简化方法的调用方式。但是在 Java 7 之前，如果可变长度的参数与泛型一起使用后会遇到一个麻烦，就是便一起产生的警告过多，比如下面 Java 6 变长参数示例。

```

public static <T>T useVarargs(T...args){
    return args.length > 0?args[0]:null;
}

```

如果参数传递的是不可具体化(non-reifiable)的类型，如 `List<String>` 这样的泛型类型，会产生警告信息。每一次调用该方法，都会产生警告信息。比如在 Java 7 之前的编译器上编译代码 `VarargsWarning useVarargs(new ArrayList<String>())`，编译器会给出警告信息。如果希望禁止这个警告信息，需要使用 `@SuppressWarnings("unchecked")` 注解来声明。这其中的原因是可变长度的方法参数的实际值是通过数组来传递的，而数组中存储的是不可具体化的泛型类对象，自身存在类型安全问题。因此编译器会给出相应的警告信息。这样的警告信息在使用 Java 标准类库中的 `java.util.Arrays` 类的 `asList` 和 `java.util.Collections` 类的 `addAll` 方法中也会遇到。建议开发人员每次使用方法时都抑制编译器的警告信息，这个不是一个好主意。

为了解决这个问题，Java 7 引入了一个新的注解 `@SafeVarargs`。如果开发人员确信某个使用了可变长度参数的方法，在与泛型类一起使用时不会出现类型安全问题，就可以用这个注解进行声明。在使用了这个注解之后，编译器遇到类似的问题，就不会再给出相关的警告信息。

`@SafeVarargs` 注解只能用在参数长度可变的方法或构造方法上，且方法必须声明为 `static` 或 `final`，否则会出现编译错误。一个方法是用 `@SafeVarargs` 注解的前提是，开发人员必须确保这个方法的实现中对泛型类型参数的处理不会引发类型安全问题。

3.1.20 针对基本数据类型的优化

Java7 对基本类型的包装类做了一些更新，以更好地满足日常的开发需求。第一个更新是在基本类型的比较方面，`Boolean`、`Byte`、`Short`、`Integer`、`Long` 和 `Character` 类都添加了一个比较两个基本类型值的静态 `compare` 方法，比如 `Long` 类的 `compare` 方法可以用来比较两个 `Long` 类型的值。这个 `compare` 方法只能简化进行基本类型数值比较时的代码。在 Java 7 之前，如果需要对两个 `int` 数值 `x` 和 `y` 进行比较，一般的做法是使用代码 `"Integer.value(x).compareTo(Integer.value(y))"`，而在 Java7 直接使用 `"Integer.compare(x,y)"`。

字符串内部化 (string interning) 技术可以提高字符串比较时的性能，是一种典型的空间换时间的做法。在 Java 中包含相同字符的字符串字面量引用的是相同的内部对象。`String` 类也提供了 `intern` 方法来返回与当前字符串内容相同的但已经包含在内部缓存中的对象引用。在对被内部缓存的字符串进行比较时，可以直接使用 `"=="` 操作符，而不需要用更加耗时的 `equals` 方法。

Java7 把这种内部化机制扩大到了 -128~127 的数字。根据 Java 语言规范，对于 -128 到 127 范

范围内的 short 类型和 int 类型，以及\u0000 到\u007f 范围内的 char 类型，它们对应的包装类对象始终指向相同的对象，即通过 “==” 进行判断时的结果为 true。为了满足这个要求，Byte、Short、Integer 类的 valueOf 方法对于 -128 到 127 范围内的值，以及 Character 类的 valueOf 方法对于 0 到 127 范围内的值，都会返回内部缓存的对象。如果希望缓存更多的值，可以通过 Java 虚拟机启动参数 “-Djava.lang.Integer.IntegerCache.high” 来进行设置。例如，使用 “-Djava.lang.Integer.IntegerCache.high=256” 之后，数值缓存的范围就变成了 -128 到 256。

3.1.21 空变量

显式地赋空变量是否有助于程序的性能。赋空变量是指简单地将 null 值显式地赋值给这个变量，相对于让该变量的引用失去其作用域。

代码清单 3-31 局部作用域

```
public static String scopingExample(String string) {
    StringBuffer sb = new StringBuffer();
    sb.append("hello ").append(string);
    sb.append(", nice to see you!");
    return sb.toString();
}
```

如清单 3-31 所示，当该方法执行时，运行时栈保留了一个对 StringBuffer 对象的引用，这个对象是在程序的第一行产生的。在这个方法的整个执行期间，栈保存的这个对象引用将会防止该对象被当作垃圾。当这个方法执行完毕，变量 sb 也就失去了它的作用域，相应地运行时栈就会删除对该 StringBuffer 对象的引用。于是不再有对该 StringBuffer 对象的引用，现在它就可以被当作垃圾收集了。栈删除引用的操作就等于在该方法结束时将 null 值赋给变量 sb。

既然 Java 虚拟机可以执行等价于赋空的操作，那么显式地赋空变量还有什么用呢？对于在正确的作用域中的变量来说，显式地赋空变量的确没用。但是让我们来看看另外一个版本的 scopingExample 方法，如代码清单 3-32 所示，这一次我们将把变量 sb 放在一个错误的作用域中。

代码清单 3-32 静态作用域

```
static StringBuffer sb = new StringBuffer();
public static String scopingExample(String string) {
    sb = new StringBuffer();
    sb.append("hello ").append(string);
    sb.append(", nice to see you!");
    return sb.toString();
}
```

现在 sb 是一个静态变量，所以只要它所在的类还装载在 Java 虚拟机中，它也将一直存在。该方法执行一次，一个新的 StringBuffer 将被创建并且被 sb 变量引用。在这种情况下，sb 变量以前引用的 StringBuffer 对象将会死亡，成为垃圾收集的对象。也就是说，这个死亡的 StringBuffer 对象被程序保留的时间比它实际需要保留的时间长得多，如果再也没有对该 scopingExample 方法的调用，它将会永远保留下去。

即使如此，显式地赋空变量能够提高性能吗？我们会发现我们很难相信一个对象会或多或少

对程序的性能产生很大影响，直到清单 3-33 所示，它包含了一个大型对象。

代码清单 3-33 仍在静态作用域中的对象

```
private static Object bigObject;

public static void test(int size) {
    long startTime = System.currentTimeMillis();
    long numObjects = 0;
    while (true) {
        //bigObject = null; //explicit nulling
        //SizableObject could simply be a large array, e.g. byte[]
        //In the JavaGaming discussion it was a BufferedImage
        bigObject = new SizableObject(size);
        long endTime = System.currentTimeMillis();
        ++numObjects;
        // We print stats for every two seconds
        if (endTime - startTime >= 2000) {
            System.out.println("Objects created per 2 seconds = " + numObjects);
            startTime = endTime;
            numObjects = 0;
        }
    }
}
```

这个例子有个简单的循环，创建一个大型对象并且将它赋给同一个变量，每隔两秒钟报告一次所创建的对象个数。现在的 Java 虚拟机采用 generational 垃圾收集机制，新的对象创建之后放在一个内存空间（取名 Eden）内，然后将那些在第一次垃圾收集以后仍然保留的对象转移到另外一个内存空间。在 Eden，即创建新对象时所在的新一代空间中，收集对象要比在“老一代”空间中快得多。但是如果 Eden 空间已经满了，没有空间可供分配，那么就必须把 Eden 中的对象转移到老一代空间中，腾出空间来给新创建的对象。如果没有显式地赋空变量，而且所创建的对象足够大，那么 Eden 就会填满，并且垃圾收集器就不能收集当前所引用的这个大型对象。所产生的后果是，这个大型对象被转移到“老一代空间”，并且要花更多的时间来收集它。

通过显式地赋空变量，Eden 就能在新对象创建之前获得自由空间，这样垃圾收集就会更快。实际上，在显式赋空的情况下，该循环在两秒钟内创建的对象个数是没有显式赋空时的 5 倍——但是仅当您选择创建的对象要足够大而可以填满 Eden 时才是如此，在 Windows 环境、Java 虚拟机 1.4 的默认配置下大概需要 500KB。那就是一行赋空操作产生的 5 倍的性能差距。但是请注意这个性能差别产生的原因是变量的作用域不正确，这正是赋空操作发挥作用的地方，并且是因为所创建的对象非常大。

3.2 集合类概念

Java Collection 类为程序员的生产力带来了巨大的提升，其提供的接口容器，可以方便地在各种具体实现之间切换。

Collection 类的某些具体实现基于数组,即由于底层数据存储基于数组,随着元素数量的增加,调整大小的代价很大,典型的代表如 ArrayList、Vector、HashMap 及 ConcurrentHashMap。另一些 Collection 类的实现,如 LinkedList 或 TreeMap,常使用一个或多个对象引用,将 Collection 类管理的各个元素串接起来。这些 Collection 类实现中的前者,使用数组作为底层的数据存储,随着 Collection 元素增长到某个上限,需要调整其大小时很容易出现性能问题。虽然这些 Collection 类也含有构造函数,可以接收优化的参数值作为 Collection 的大小,但构造函数并不经常被使用,或者应用程序中提供的大小并没有针对该 Collection 类做优化。

以 StringBuilder/StringBuffer 为例,使用数组作为数据存储的 Java Collection 类需要消耗额外的 CPU 周期分配新数组,将老的元素从旧数组中复制到新数组中,在将来的某个时刻还需要对数组进行垃圾收集。此外,调整大小还会影响 Collection 类字段的访问时间及解析引用字段的时间,因为作为一个新的底层数据存储(典型的即为数组),它可能被分配到 JVM 堆中的某个位置,与 Collection 类中其他的字段及对象引用不在同一块内存存储。Collection 类发生大小调整后,访问调整后的字段可能会导致 CPU 高速缓存未命中,这是由现代 JVM 在内存中分配对象的方式导致的,尤其是对象在内存中如何分布决定的。不同的 Java 虚拟机实现中,对象及其字段在内存中的分布可能有所不同。然而,一般来说,由于对象及其字段常常需要同时引用,将对象及其字段尽可能放在内存中相邻的位置能够减少 CPU 高速缓存未命中。由此可见,Collection 类大小调整的影响已经远远不止大小调整所额外消耗的 CPU 指令,还会对 JVM 的内存管理器造成影响,由于改变了内存中 Collection 类字段相对于对象实例的布局,字段的访问时间将会变长。本节要讲解针对集合相关数据结构使用的优化建议。

3.2.1 快速删除 List 里面的数据

java.util.List 的 subList 方法可以被用来返回一个集合(List)的一部分的数据。

List<E> subList(int fromIndex, int toIndex),该方法定义返回原有集合的从 fromIndex 到 toIndex 之间这一部分的数据,组成一个新的集合,这两个集合之间是有关联的。所以,你对原来的 list 和返回的 list 做的非结构性修改³,都会影响到彼此对方。如果发生结构性修改的是原来的 list (不包括由于返回的子 list 导致的改变),那么返回的子 list 语义上将会是 undefined。在 AbstractList 中,undefined 的具体表现形式是抛出一个 ConcurrentModificationException。

因此,如果你在调用了 sublist 返回了子 list 之后,假设这个时候你又修改了原 list 的大小,那么之前产生的子 list 就会失效,即子 list 会变成不可用状态。相应地,如果你想要修改原有的 list,你也可以通过调用 sublist 的方式来实现这个需求,即调用代码 list.subList(from, to).clear()。

如清单 3-34 所示,通过对返回的队列、原队列进行修改,也会触发对应的变化。

代码清单 3-34 sublist 示例

```
import java.util.ArrayList;
```

```
import java.util.List;
```

³ 即 non-structural changes,是指不涉及到 list 的大小改变的修改。相反,结构性修改,指改变了 list 大小的修改。

```

public class testSubList {

    public static void main(String[] args) {
        List<String> parentList = new ArrayList<String>();

        for(int i = 0; i < 5; i++){
            parentList.add(String.valueOf(i));
        }

        List<String> subList = parentList.subList(1, 3);
        for(String s : subList){
            System.out.println(s);//output: 1, 2
        }

        System.out.println("-----");
        //non-structural modification by sublist, reflect parentList
        subList.set(0, "new 1");
        for(String s : parentList){
            System.out.println(s);//output: 0, new 1, 2, 3, 4
        }

        System.out.println("-----");
        //structural modification by sublist, reflect parentList
        subList.add(String.valueOf(2.5));
        for(String s : parentList){
            System.out.println(s);//output:0, new 1, 2, 2.5, 3, 4
        }

        System.out.println("-----");
        //non-structural modification by parentList, reflect sublist
        parentList.set(2, "new 2");
        for(String s : subList){
            System.out.println(s);//output: new 1, new 2
        }

        System.out.println("-----");
        //structural modification by parentList, sublist becomes undefined(throw
exception)
        parentList.add("undefine");
        for(String s : subList){
            System.out.println(s);
        }

        System.out.println("-----");
        subList.get(0);
    }
}

```

清单 3-34 程序运行之后的输出如清单 3-35 所示。

代码清单 3-35 运行程序输出

1
2


```
-----
0
new 1
2
3
4
-----
0
new 1
2
2.5
3
4
-----
new 1
new 2
2.5
-----
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.ArrayList$SubList.checkForComodification(Unknown Source)
    at java.util.ArrayList$SubList.listIterator(Unknown Source)
    at java.util.AbstractList.listIterator(Unknown Source)
    at java.util.ArrayList$SubList.iterator(Unknown Source)
    at testSubString.main(testSubString.java:38)
```

3.2.2 集合内部避免返回 null

当我们在需要返回数组或者集合的方法中，如果需要返回空数据，则不要返回 `null`，而是要返回大小为 0 的数组或者集合。

如代码清单 3-36 所示代码，当队列大小为 0 时，返回 `Null`。

代码清单 3-36 集合返回空示例

```
private final List<String> bookInStock;

/**
 * @return an array containing all of the book in the shop,
 * or null if no book are available for purchase. */
public String[] getBooks() {
    if (bookInStock.size() == 0)
        return null;
}

}
```

采用这种返回 `Null` 的方式，会导致一个问题，就是调用这个集合时必须判断这个方法返回的是否为 `null`，否则，可能会抛出 `NullPointerException` 异常，如代码清单 3-37 所示。

代码清单 3-37 判断 Null

```
Book[] books = shop.getBooks();
```

```

if (books != null && Arrays.asList(books).contains(books.STILTON)){
    System.out.println("Sorry, Empty!.");
}

```

坚持选择返回 `null` 的原因可能是认为分配一个空数组或者集合是需要花费时间和空间的，这样会间接影响性能。但是总的来说，这点性能损耗很小，此外，返回的空数组或者集合通常是 `immutable`，即不可变的⁴，所以可以定义成 `static final`（对于数组而言）或者 `Collections.emptyList()/emptyMap()/emptySet()` 来公用同一个对象，减少性能影响。实现方式如清单 3-38 所示。

代码清单 3-38 正确的返回方式

```

// The right way to return an array from a collection
private final List<Book> bookInStock = ...;
private static final Book [] EMPTY_BOOK_ARRAY = new Book[0];
/** @return an array containing all of the book in the shop.
 */
public Book[] getBook() {
    return bookInStock.toArray(EMPTY_BOOK_ARRAY);
}
// The right way to return a copy of a collection
public List<Book> getBookList() {
    if (bookInStock.isEmpty()){
        return Collections.emptyList();
    }
    // Always returns same list
} else {
    return new ArrayList<Book>(bookInStock);
}
}

```

结合上面的代码及描述信息，总的来说，对于集合可能为空的应用程序我们需要注意以下几点：

- (1) 自己写接口方法时尽量遵守这条规范，并在方法注释中标明，尽量返回大小为 0 的数组或者集合。
- (2) 文中提到的 `Collections.emptyList()` 这个方法需要慎用。因为这个方法返回的集合对象都是 `immutable` 的，即不可更改的。而有时候我们可能需要向返回的集合中添加或者删除元素，这样的话，就会触发 `UnsupportedOperationException` 异常。
- (3) 当要判断某个元素是否在数组中时，可以采用 `Arrays.asList(T[] array).contains(T obj)` 来作为实现方案，而无须用循环迭代每个元素来判断。
- (4) 如果要把 `ArrayList` 变成数组，可以使用 `ArrayList.toArray(T[] array)`，参数里的 `array` 只需设成大小为 0 的数组即可，仅用来指定返回的类型，`T` 可能为 `ArrayList` 中元素的子类。

⁴ `String` 不可变的原因包括设计考虑、效率优化问题，以及安全性这三大方面。

3.2.3 ArrayList、LinkedList 比较

ArrayList、Vector、LinkedList 均来自抽象类 AbstractList 的实现，而 AbstractList 扩展自 AbstractCollection，并直接实现了 List 接口。AbstractList 最大限度地减少了实现由“随机访问”数据存储（如数组）支持的接口所需的工作。需要注意的是，对于连续的访问数据（如链表），应优先使用 AbstractSequentialList，这个抽象类最大限度地减少了实现受“连续访问”数据存储（如链接列表）支持的此接口所需的工作。

ArrayList 和 Vector 使用数组原理实现，其中 ArrayList 没有对任何一个方法做线程同步保护，因此整体上来说，ArrayList 不是线程安全的。Vector 中绝大部分方法都做了线程同步，所以可以说它是线程安全的实现。LinkedList 使用了循环双向链表数据结构，由一系列表项连接而成，一个表项由 3 个部分组成，即元素内容、前驱表项和后驱表项。

当 ArrayList 对容量的需求超过当前数组预定义的最大限时，数组需要进行扩容，扩容过程中会进行大量的数组复制操作，而对于数组复制操作来说，最终将调用 System.arraycopy() 方法。LinkedList 由于使用了链表的结构，因此不需要维护容量的大小，然而每次新增元素时都需要新建一个 Entry 对象，并进行更多的赋值操作，因此在频繁的系统调用下，容易对性能产生一定的影响，在不间断地生成新的对象过程中占用了一定的资源。

前面说过，ArrayList 是基于数组实现的，数组是一块连续的内存空间，因为数组的连续性，因此总是在尾端增加元素，只有在空间不足时才产生数组扩容和数组复制。如果在数组的任意位置插入元素，必然导致在该位置后的所有元素需要重新排列，因此其效率较差，尽可能将数据插入到尾部。LinkedList 由于基于链表数据结构，因此不会因为插入数据而导致性能下降。

总的来说，ArrayList 允许所有元素，可以包括 Null 元素，并可以根据索引位置对集合进行快速的随机访问，缺点是向指定的索引位置插入对象或删除对象的速度较慢。ArrayList 的每一次有效的元素删除操作后都要进行数组的重组，并且删除的元素位置越靠前，数组重组时的开销越大，要删除的元素位置越靠后，开销越小。LinkedList 要移除中间的数据需要便利完半个 List。

清单 3-39 所示代码演示了 ArrayList 和 LinkedList 的操作方式。

代码清单 3-39 ArrayList 和 LinkedList 示例代码

```
import java.util.ArrayList;

import java.util.LinkedList;

public class ArrayListandLinkedList {
    public static void main(String[] args){
        long start = System.currentTimeMillis();
        ArrayList list = new ArrayList();
        Object obj = new Object();
        for(int i=0;i<5000000;i++){
            list.add(obj);
        }
        long end = System.currentTimeMillis();
        System.out.println(end-start);

        start = System.currentTimeMillis();
```

```

LinkedList list1 = new LinkedList();
Object obj1 = new Object();
for(int i=0;i<5000000;i++){
    list1.add(obj1);
}
end = System.currentTimeMillis();
System.out.println(end-start);

start = System.currentTimeMillis();
Object obj2 = new Object();
for(int i=0;i<1000;i++){
    list.add(0,obj2);
}
end = System.currentTimeMillis();
System.out.println(end-start);

start = System.currentTimeMillis();
Object obj3 = new Object();
for(int i=0;i<1000;i++){
    list1.add(obj1);
}
end = System.currentTimeMillis();
System.out.println(end-start);

start = System.currentTimeMillis();
list.remove(0);
end = System.currentTimeMillis();
System.out.println(end-start);

start = System.currentTimeMillis();
list1.remove(250000);
end = System.currentTimeMillis();
System.out.println(end-start);

```

程序对 ArrayList 和 LinkedList 分别执行插入数据、删除数据操作，输出各个操作所消耗的时间，单位为秒。本实验中，第一次插入数据时，ArrayList 插入 500 万元素时比同样操作方式下的 LinkedList 要快将近 2 倍。但是当初始化成功后，继续向 ArrayList 新增 1000 个元素时，速度较同样情况下的 LinkedList 慢很多。删除数据的时间 ArrayList 比 LinkedList 要快。具体运行输出如清单 3-40 所示。

代码清单 3-40 清单 3-39 运行输出

639

1296

6969
0
0
15

从上面的实验我们可以得出，如果数据较为稳定，不容易发生新增操作，那么我们可以考虑使用 ArrayList。如果容易发生修改，例如互联网应用场景，应用面向的是大众用户平台，元素的修改较为频繁，那么我们可以考虑使用 LinkedList。

LinkedList 采用的数据结构导致了它的排序方式不同，也导致了它获取数据的方式不同。假设我们的程序顺序访问一个 LinkedList 的元素，我们会发现随着 index 的变大，程序速度会越来越慢，list 的元素个数在百万以上。下面程序中错误的代码就是使用了 list.get(i)，注意顺序访问，LinkedList 绝对不要用 get 方法，即使 LinkedList 的元素个数只有很少的几个。发生这个情况的原因是 LinkedList 的底层是一个链表，随机访问 i 的时候，链表只能从前往后数，第 i 个才返回。所以时间随着 i 的变大时间会越来越长。程序代码如代码清单 3-41 所示。

代码清单 3-41 LinkedList 错误的使用方式

```
import java.util.LinkedList;

import java.util.List;

public class LinkedListWrongDemo {
    public static void main(String[] args) {
        // add elements
        int size = 2000000;
        List<String> list = new LinkedList<String>();
        for (int i = 0; i < size; i++) {
            list.add("Just some test data");
        }

        long startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            list.get(i);
            if (i % 10000 == 0) {
                System.out.println("query 10000 elements spend: "+
                    (System.currentTimeMillis() - startTime));
                startTime = System.currentTimeMillis();
            }
        }
    }
}
```

代码清单 3-41 所示的代码运行后打印输出如代码清单 3-42 所示。

代码清单 3-42 3-41 代码运行输出

query 10000 elements spend: 0

```
query 10000 elements spend: 375
query 10000 elements spend: 983
query 10000 elements spend: 1639
query 10000 elements spend: 2388
query 10000 elements spend: 2981
query 10000 elements spend: 3448
query 10000 elements spend: 2653
query 10000 elements spend: 2560
query 10000 elements spend: 2949
query 10000 elements spend: 3230
```

此外，这里介绍一下 `RandomAccess` 接口。`RandomAccess` 接口是一个标志接口，它本身并没有提供任何方法，实现了 `RandomAccess` 接口的对象都可以被认为是支持快速随机访问的对象。该接口的主要目的是标识那些可支持快速随机访问的 `List` 实现。随机和连续访问之间的区别通常是模糊的。例如，如果列表很大时，某些 `List` 实现提供渐进的线性访问时间，但实际上是固定的访问时间。这样的 `List` 实现通常应该实现此接口。

任何一个基于数组的 `List` 实现都实现了 `RandomAccess` 接口，而基于链表的实现则都没有。因为只有数组能够进行快速的随机访问，而对链表的随机访问需要进行链表的遍历。因此，此接口的好处是，可以在应用程序中知道正在处理的 `List` 对象是否可以快速随机访问，从而针对不同的 `List` 进行不同的操作，以提高程序的性能。可以使用 `list instanceof RandomAccess` 语句来判断是否是数组实现的。JDK 中说得很清楚，在对 `List` 特别是 Huge size 的 `List` 的遍历算法中，要尽量来判断是属于 `RandomAccess`（如 `ArrayList`）还是 `Sequence List`（如 `LinkedList`），因为适合 `RandomAccess List` 的遍历算法，用在 `Sequence List` 上就差别很大，常用的做法就是要作一个判断，如代码清单 3-43 所示。

代码清单 3-43 判断是否适合 `RandomAccess`

```
if (list instanceof RandomAccess){
    for(int m = 0; m < list.size(); m++){
    }
}else{
    Iterator iter = list.iterator();
    while(iter.hasNext()){
    }
}
```

3.2.4 Vector、HashTable 比较

`Vector` 是 `java.util` 包的类，他的功能是实现了一个动态增长的数组，像其他数组一样，此向量数组可以为每个包含的元素分配一下整数索引号，但是，向量不同于数组，它的长度可以在创建以后根据实际包含的元素个数增加或减少。容量因素的值总是大于向量的长度，因为当元素被添加到向量中，向量存储长度的增加是以增长幅度因素指定的值来增加的，应用程序可以在插入大量元素前，先根据需要增加适量的向量容量，这样，可以避免增加多余的存储空间。

`Hashtable` 类实现一个哈希表，该哈希表将键映射到相应的值。任何非 `null` 对象都可以用作键或值。为了成功地在哈希表中存储和获取对象，用作键的对象必须实现 `hashCode` 方法和 `equals` 方法。

Hashtable 的实例有两个参数影响其性能,即初始容量和加载因子。容量是哈希表中桶的数量,初始容量就是哈希表创建时的容量。注意,哈希表的状态为 open,表示在发生“哈希冲突”的情况下,单个桶会存储多个条目,这些条目必须按顺序搜索。加载因子是对哈希表在其容量自动增加之前可以达到多满的一个尺度。初始容量和加载因子这两个参数只是对该实现的提示。关于何时以及是否调用 rehash 方法的具体细节则依赖于该实现。通常,默认加载因子(0.75)在时间和空间成本上寻求一种折中。加载因子过高虽然减少了空间开销,但同时也增加了查找某个条目的时间(在大多数 Hashtable 操作中,包括 get 和 put 操作,都反映了这一点)。

初始容量主要控制空间消耗与执行 rehash 操作所需要的时间损耗之间的平衡。如果初始容量大于 Hashtable 所包含的最大条目数除以加载因子,则永远不会发生 rehash 操作。但是,将初始容量设置太高可能会浪费空间。如果很多条目要存储在一个 Hashtable 中,那么与根据需求执行自动 rehashing 操作来增大表的容量的做法相比,使用足够大的初始容量创建哈希表或许可以更有效地插入条目。

使用 Vector 或者 HashTable 的时候,我们可以为 Vector 和 HashTable 定义初始大小,我们知道,Vector 来自抽象类 AbstractList 的实现,并且 Vector 是基于数组实现的,那么 JVM 为 Vector 扩充大小的时候需要重新创建一个更大的数组,首先将原先数组中的内容复制过来,然后原先的数组才会被回收。可见 Vector 容量的扩大是一个颇费时间的事。通常,默认的 10 个元素大小是不够的,读者最好能根据实际应用场景准确的估计你所需要的最佳大小。

代码如清单 3-44 所示,由于事先没有确定 Vector 数组元素个数,所以每次新增元素时都需要执行复制操作,较为耗时。

代码清单 3-44 Vector 示例代码

```
import java.util.vector;  
public class VectorDemo{  
    public void addobjects (object[] o) {  
        // if length > 10, vector needs to expand  
        for (int i = 0; i< o.length;i++) {  
            v.add(o); // capacity before it can add more elements.  
        }  
    }  
    public vector v = new vector(); // no initialcapacity.  
}
```

我们可以通过 `public vector v = new vector(20);` `public hashtable hash = new hashtable(10);` 这样的方式自己设定初始大小。

需要读者注意,单线程应尽量使用 HashMap、ArrayList,除非必要,否则不推荐使用 Hashtable、Vector,这两个类使用了同步机制,反而会降低性能。JDK 的发展过程是,用 ArrayList 代替 Vector。Vector 是线程安全的,而有的时候我们确实希望在多线程的情况下使用列表,那么这个时候我们可以利用 Collections 这个类当中为我们提供的 `synchronizedList(List list)`,它可以返回一个线程安全的同步的列表,还提供了返回同步的 Collections。用 HashMap 代替 Hashtable。Hashtable 是线程安全的,而有的时候我们确实希望在多线程的情况下使用 HashMap,那么这个时候我们可以利用 Collections 这个类当中为我们提供的 `synchronizedMap(Map<K,V> m)`,它可以返回一个线程安全的

同步的 `HashMap`。用 `LinkedList` 代替 `Stack`。当初在设计 `Stack` 的时候就有一些潜在的问题，它是从 `Vector` 继承而来，对于一个栈来说，它只能是最后放进去的元素，要先出来，但是它继承自 `Vector`，而 `Vector` 中有一个方法叫作 `elementAt(int index)`，而不能说是通过这个索引 `index` 去任意的获得一个元素。结果它就有了这个奇怪的特性，提倡应该自己利用 `LinkedList` 去实现一个 `stack`。

3.2.5 HashMap 使用经验

小时候妈妈都教过我们，不同的东西要放在不同的位置，需要时才能快速找到它。当然这个规则你必须记住，不然怎么都找不到了。`HashMap` 之所以能够做到快速存、取，与我们下面要介绍的内容密切相关。

`HashMap` 和 `HashSet` 是 Java Collection Framework 的两个重要成员，其中 `HashMap` 是 `Map` 接口的常用实现类，`HashSet` 是 `Set` 接口的常用实现类。虽然 `HashMap` 和 `HashSet` 实现的接口规范不同，但它们底层的 Hash 存储机制完全一样，甚至 `HashSet` 本身就采用 `HashMap` 来实现的。读者请注意，虽然集合号称存储的是 Java 对象，但实际上并不会真正将 Java 对象放入 `Set` 集合中，只是在 `Set` 集合中保留这些对象的引用而言。也就是说：Java 集合实际上是多个引用变量所组成的集合，这些引用变量指向实际的 Java 对象。就像引用类型的数组一样，当我们把 Java 对象放入数组之时，并不是真正把 Java 对象放入数组中，只是把对象的引用放入数组中，每个数组元素都是一个引用变量。

我们首先来谈谈 `HashMap`。`HashMap` 是基于哈希表的 `Map` 接口的实现，`HashMap` 将 Hash 值映射到内存地址，直接取得 Key 所对应的数据。在 `HashMap` 中，底层数据结构使用的是数组，所谓的内存地址即数组的下标索引。总的来说，除了非同步和允许使用 `Null`，`HashMap` 类与 `Hashtable` 大致相同。

`HashMap` 实际上是一个链表的数组。基于 `HashMap` 的链表方式实现机制，只要 `hashCode()` 和 `hash()` 方法实现得足够好，能够尽可能地减少冲突的产生，那么对 `HashMap` 的操作几乎等价于对数组的随机访问操作，具有很好的性能。但是，如果 `hashCode()` 或者 `hash()` 方法实现较差，在大量冲突产生的情况下，那么 `HashMap` 事实上就退化为几个链表，对 `HashMap` 的操作等价于遍历链表，此时性能很差。

`HashMap` 的一个功能缺点是它的无序性，即被存入到 `HashMap` 中的元素，在遍历 `HashMap` 时，其输出是无序的。如果希望元素保持输入的顺序，可以使用 `LinkedHashMap` 替代。

`HashMap` 采用一种所谓的“Hash 算法”来决定每个元素的存储位置。

当程序试图将多个 key-value 放入 `HashMap` 中时，如代码清单 3-45 所示。

代码清单 3-45 HashMap 初始化示例

```
HashMap<String , Double> map = new HashMap<String , Double>();
map.put("xiao ming" , 80.0);
map.put("xiao hong" , 89.0);
map.put("xiao hua" , 78.2);
```

当程序执行 `map.put("xiao ming",80.0);` 时，系统将调用 `"xiao ming"` 的 `hashCode()` 方法得到其 `hashCode` 值——每个 Java 对象都有 `hashCode()` 方法，都可通过该方法获得它的 `hashCode` 值。当我们得到这个对象的 `hashCode` 值之后，系统会根据该 `hashCode` 值来决定该元素的存储位置。

我们来看一下 HashMap 源代码中的 put(K key,V value)方式, 代码如清单 3-46 所示。

代码清单 3-46 HashMap 源代码

```
public V put(K key, V value)
{
    // 如果 key 为 null, 调用 putForNullKey 方法进行处理
    if (key == null)
        return putForNullKey(value);
    // 根据 key 的 hashCode 计算 Hash 值
    int hash = hash(key.hashCode());
    // 搜索指定 hash 值在对应 table 中的索引
    int i = indexFor(hash, table.length);
    // 如果 i 索引处的 Entry 不为 null, 通过循环不断遍历 e 元素的下一个元素
    for (Entry<K,V> e = table[i]; e != null; e = e.next)
    {
        Object k;
        // 找到指定 key 与需要放入的 key 相等 (hash 值相同)
        // 通过 equals 比较放回 true)
        if (e.hash == hash && ((k = e.key) == key
            || key.equals(k)))
        {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    // 如果 i 索引处的 Entry 为 null, 表明此处还没有 Entry
    modCount++;
    // 将 key、value 添加到 i 索引处
    addEntry(hash, key, value, i);
    return null;
}
```

清单 3-46 代码调用了一个 hash 方法用于生成 hash 值, 它是一个纯粹的数学方法, 源代码如清单 3-47 所示。

代码清单 3-47 HashMap 的 hash 方法源代码

```
static int hash(int h)
{
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

对于任意给定的对象, 只要它的 hashCode()返回值相同, 那么程序调用 hash(int h)方法所计算得到的 Hash 码值总是相同的。接下来程序会调用 indexFor(int h, int length)方法来计算该对象应该保存在 table 数组的哪个索引处。indexFor(int h, int length)方法的代码如清单 3-48 所示。

代码清单 3-48 HashMap 的 indexFor 方法源代码

```
static int indexFor(int h, int length)
{
    return h & (length-1);
}
```

这个方法非常巧妙，它总是通过 $h \& (table.length-1)$ 来得到该对象的保存位置，而 HashMap 底层数组的长度总是 2 的 n 次方，当 $length$ 总是 2 的倍数时， $h \& (length-1)$ 是一个非常巧妙的设计：假设 $h=5$, $length=16$ ，那么 $h \& length-1$ 将得到 5；如果 $h=6$, $length=16$ ，那么 $h \& length-1$ 将得到 6，如果 $h=15$, $length=16$ ，那么 $h \& length-1$ 将得到 15；但是当 $h=16$ 时， $length=16$ 时，那么 $h \& length-1$ 将得到 0 了；当 $h=17$ 时， $length=16$ 时，那么 $h \& length-1$ 是 1，这样保证计算得到的索引值总是位于 $table$ 数组的索引之内。

根据上面 3-46 所示的 `put` 方法源代码可以看出，当程序试图将一个 `key-value` 对放入 HashMap 中时，程序首先根据该 `key` 的 `hashCode()` 返回值决定该 `Entry` 的存储位置，如果两个 `Entry` 的 `key` 的 `hashCode()` 返回值相同，那它们的存储位置相同。如果这两个 `Entry` 的 `key` 通过 `equals` 比较返回 `true`，新添加 `Entry` 的 `Value` 将覆盖集中原有 `Entry` 的 `Value`，但 `key` 不会被覆盖。如果这两个 `Entry` 的 `key` 通过 `equals` 比较返回 `false`，新添加的 `Entry` 将与集中原有 `Entry` 形成 `Entry` 链，而且新添加的 `Entry` 位于 `Entry` 链的头部。

当向 HashMap 中添加 `key-value` 对，由其 `key` 的 `hashCode()` 返回值决定该 `key-value` 对（就是 `Entry` 对象）的存储位置。当两个 `Entry` 对象的 `key` 的 `hashCode()` 返回值相同时，将由 `key` 通过 `equals()` 比较值决定是采用覆盖行为（返回 `true`），还是产生 `Entry` 链（返回 `false`）。

清单 3-46 所示代码中也调用了 `addEntry(hash, key, value, i);` 方法，`addEntry` 是 HashMap 提供的一个包访问权限的方法，该方法仅用于添加一个 `key-value` 对。代码如清单 3-49 所示。

代码清单 3-49 HashMap 的 addEntry 方法源代码

```
void addEntry(int hash, K key, V value, int bucketIndex)
{
    // 获取指定 bucketIndex 索引处的 Entry
    // table 是一个普通数组，每个数组都有一个固定的长度，这个数组的长度就是 HashMap 的容量。
    Entry<K,V> e = table[bucketIndex]; //
    // 将新创建的 Entry 放入 bucketIndex 索引处，并让新的 Entry 指向原来的 Entry
    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
    // 如果 Map 中的 key-value 对的数量超过了极限
    // Size 变量用于保存该 HashMap 中所包含的 key-value 对的数量。
    // threshold 变量包含了 HashMap 能容纳的 key-value 对的极限，它的值等于 HashMap 的容量乘以负载因子 (load factor)。
    // 当 size++ >= threshold 时，HashMap 会自动调用 resize 方法扩充 HashMap 的容量。每扩充一次，HashMap 的容量就增大一倍。
    if (size++ >= threshold)
        // 把 table 对象的长度扩充到 2 倍
        resize(2 * table.length);
}
```

系统总是将新添加的 `Entry` 对象放入 $table$ 数组的 `bucketIndex` 索引处，如果 `bucketIndex` 索引

处已经有了一个 Entry 对象，那新添加的 Entry 对象指向原有的 Entry 对象（产生一个 Entry 链），如果 bucketIndex 索引处没有 Entry 对象，那么通过代码 `Entry<K,V> e = table[bucketIndex]`；确保 e 变量是 null，也就是新放入的 Entry 对象指向 Null，也就是没有产生 Entry 链。

当 HashMap 的每个 bucket 里存储的 Entry 只是单个 Entry，也就是没有通过指针产生 Entry 链时，此时的 HashMap 具有最好的性能。当程序通过 key 取出对应 value 时，系统只要先计算出该 key 的 hashCode() 返回值，再根据该 hashCode 返回值找出该 key 在 table 数组中的索引，然后取出该索引处的 Entry，最后返回该 key 对应的 value 即可。HashMap 类的 get(K key) 方法代码如清单 3-50 所示。

代码清单 3-50 HashMap 的 get 方法源代码

```
public V get(Object key)
{
    // 如果 key 是 null，调用 getForNullKey 取出对应的 value
    if (key == null)
        return getForNullKey();
    // 根据该 key 的 hashCode 值计算它的 hash 码
    int hash = hash(key.hashCode());
    // 直接取出 table 数组中指定索引处的值，
    for (Entry<K,V> e = table[indexFor(hash, table.length)];
        e != null;
        // 搜索该 Entry 链的下一个 Entr
        e = e.next)    // ①
    {
        Object k;
        // 如果该 Entry 的 key 与被搜索 key 相同
        if (e.hash == hash && ((k = e.key) == key
            || key.equals(k)))
            return e.value;
    }
    return null;
}
```

如果 HashMap 的每个 bucket 里只有一个 Entry 时，HashMap 可以根据索引、快速地取出该 bucket 里的 Entry。在发生“Hash 冲突”的情况下，单个 bucket 里存储的不是一个 Entry，而是一个 Entry 链，系统只能必须按顺序遍历每个 Entry，直到找到想搜索的 Entry 为止——如果恰好要搜索的 Entry 位于该 Entry 链的最末端（该 Entry 是最早放入该 bucket 中），那系统必须循环到最后才能找到该元素。

归纳起来简单地说，HashMap 在底层将 key-value 当成一个整体进行处理，这个整体就是一个 Entry 对象。HashMap 底层采用一个 Entry[] 数组来保存所有的 key-value 对，当需要存储一个 Entry 对象时，会根据 Hash 算法来决定其存储位置；当需要取出一个 Entry 时，也会根据 Hash 算法找到其存储位置，直接取出该 Entry。

当创建 HashMap 时，有一个默认的负载因子（load factor），其默认值为 0.75，这是时间和空间成本上一种折中，增大负载因子可以减少 Hash 表（就是那个 Entry 数组）所占用的内存空间，

但会增加查询数据的时间开销，而查询是最频繁的操作（HashMap 的 `get()` 与 `put()` 方法都要用到查询）；减小负载因子会提高数据查询的性能，但会增加 Hash 表所占用的内存空间。

综上所述，我们可以在创建 HashMap 时根据实际需要适当地调整 load factor 的值，如果程序比较关心空间开销、内存比较紧张，可以适当地增加负载因子，如果程序比较关心时间开销，内存比较宽裕则可以适当的减少负载因子。通常情况下，程序员无须改变负载因子的值。

如果开始就知道 HashMap 会保存多个 key-value 对，可以在创建时就使用较大的初始化容量，如果 HashMap 中 Entry 的数量一直不会超过极限容量（`capacity * load factor`），HashMap 就无须调用 `resize()` 方法重新分配 table 数组，从而保证较好的性能。当然，开始就将初始容量设置太高可能会浪费空间（系统需要创建一个长度为 capacity 的 Entry 数组），因此创建 HashMap 时初始化容量设置也需要小心对待。

从上面的源代码分析可以得出，HashMap 的高性能需要以下 3 点来提供保证。

- （1）提供高效的 Hash 算法；
- （2）提供高效的算法，保证 Hash 值到内存地址（数组索引）的映射速度；
- （3）根据内存地址（数组索引）可以直接取得对应的值。

此外，能够不用 Map 就不要用了吧，当我们想遍历一个用键值对形式保存的 Map 时，下面两种方式其实效率都不高，如清单 3-51 所示。

代码清单 3-51 map 循环代码

```
for (K key : map.keySet()) {
    V value : map.get(key);
}
for (Entry<K, V> entry : map.entrySet()) {
    K key = entry.getKey();
    V value = entry.getValue();
}
```

3.2.6 EnumSet、EnumMap

我们用到类似枚举（enum-like）结构的键值时，就应该考虑将这些键值用声明为枚举类型，并将之作为 EnumMap 键。在某些情况下，比如在使用配置 Map 时，我们可能会预先知道保存在 Map 中键值。如果这个键值非常小，我们就应该考虑使用 EnumSet 或 EnumMap，而并非使用我们常用的 HashSet 或 HashMap。

EnumSet 是一个与枚举类型一起使用的专用 Set 实现。枚举 Set 中所有元素都必须来自单个枚举类型（即必须是同类型，且该类型是 Enum 的子类）。枚举类型在创建 Set 时显式或隐式地指定。枚举 Set 在内部表示为位向量。此表示形式非常紧凑且高效。此类的时间和空间性能应该很好，足以用作传统上基于 int 的“位标志”的替换形式，具有高品质、类型安全的优势。像大多数 Collection 一样，EnumSet 是不同步的。如果指定的 Collection 也是一个枚举 Set，则批量操作（如 `containsAll` 和 `retainAll`）也应该运行得非常快。由 `iterator` 方法返回的迭代器按其自然顺序遍历这些元素（该顺序是声明枚举常量的顺序）。`iterator` 方法返回的迭代器是弱一致的，即它从不抛出

`ConcurrentModificationException` 异常，也不一定显示在迭代进行时发生的任何 `Set` 修改的效果。`EnumSet` 不允许使用 `Null` 元素，如果你试图插入 `Null` 元素将抛出 `NullPointerException` 异常。

`EnumMap` 是专门为枚举类型量身定做的 `Map` 实现。虽然使用其他的 `Map` 实现（如 `HashMap`）也能完成枚举类型实例到值得映射，但是使用 `EnumMap` 会更加高效，它只能接收同一枚举类型的实例作为键值，并且由于枚举类型实例的数量相对固定并且有限，所以 `EnumMap` 使用数组来存放与枚举类型对应的值。这使得 `EnumMap` 的效率非常高。`EnumMap` 在内部使用枚举类型的 `ordinal()` 得到当前实例的声明次序，并使用这个次序维护枚举类型实例对应值在数组的位置。在实际使用中，`EnumMap` 对象往往是由外部负责整个应用初始化的代码来填充的。

`EnumSet` 的源代码实现如清单 3-52 所示。

代码清单 3-52 EnumSet 源代码实现

```
private transient Object[] vals;

public V put(K key, V value) {
    // ...
    int index = key.ordinal();
    vals[index] = maskNull(value);
    // ...
}
```

上段代码的关键实现在于，我们用数组代替了哈希表。尤其是向 `Map` 中插入新值时，所要做的仅仅是获得一个由编译器为每个枚举类型生成的常量序列号。虽然使用其他的 `Map` 实现（如 `HashMap`）也能完成枚举类型实例到值得映射，但是使用 `EnumMap` 会更加高效。如果有一个全局的 `Map` 配置（例如只有一个实例），在增加访问速度的压力下，`EnumMap` 会获得比 `HashMap` 更加出色的性能。原因在于 `EnumMap` 使用的堆内存比 `HashMap` 要少一位（bit），而且 `HashMap` 要在每个键值上都要调用 `hashCode()` 方法和 `equals()` 方法。

注意，在不能使用 `EnumMap` 的时候，至少也要优化 `HashMap` 的 `hashCode()` 和 `equals()` 方法。一个好的 `hashCode()` 方法是很有必要的，因为它能防止对高开销 `equals()` 方法多余的调用。

3.2.7 HashSet 使用经验

总的来说，`HashSet` 和 `HashMap` 之间有很多相似之处。前面讲过，对于 `HashMap` 而言，系统把 `key-value` 当成一个整体进行处理，系统总是根据 `Hash` 算法来计算 `key-value` 的存储位置，这样可以保证能快速存、取 `Map` 的 `key-value` 对。对于 `HashSet` 而言，系统采用 `Hash` 算法决定集合元素的存储位置，这样可以保证能快速存、取集合元素。

`HashSet` 的源代码如代码清单 3-53 所示。

代码清单 3-53 HashSet 源代码

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
{
    // 使用 HashMap 的 key 保存 HashSet 中所有元素
```

```

private transient HashMap<E, Object> map;
// 定义一个虚拟的 Object 对象作为 HashMap 的 value
private static final Object PRESENT = new Object();
...
// 初始化 HashSet, 底层会初始化一个 HashMap
public HashSet()
{
    map = new HashMap<E, Object>();
}
// 以指定的 initialCapacity、loadFactor 创建 HashSet
// 其实就是以相应的参数创建 HashMap
public HashSet(int initialCapacity, float loadFactor)
{
    map = new HashMap<E, Object>(initialCapacity, loadFactor);
}
public HashSet(int initialCapacity)
{
    map = new HashMap<E, Object>(initialCapacity);
}
HashSet(int initialCapacity, float loadFactor, boolean dummy)
{
    map = new LinkedHashMap<E, Object>(initialCapacity
        , loadFactor);
}
// 调用 map 的 keySet 来返回所有的 key
public Iterator<E> iterator()
{
    return map.keySet().iterator();
}
// 调用 HashMap 的 size() 方法返回 Entry 的数量, 就得到该 Set 里元素的个数
public int size()
{
    return map.size();
}
// 调用 HashMap 的 isEmpty() 判断该 HashSet 是否为空,
// 当 HashMap 为空时, 对应的 HashSet 也为空
public boolean isEmpty()
{
    return map.isEmpty();
}
// 调用 HashMap 的 containsKey 判断是否包含指定 key
// HashSet 的所有元素就是通过 HashMap 的 key 来保存的
public boolean contains(Object o)
{
    return map.containsKey(o);
}
// 将指定元素放入 HashSet 中, 也就是将该元素作为 key 放入 HashMap
public boolean add(E e)

```



```
{
    return map.put(e, PRESENT) == null;
}
// 调用 HashMap 的 remove 方法删除指定 Entry, 也就删除了 HashSet 中对应的元素
public boolean remove(Object o)
{
    return map.remove(o) == PRESENT;
}
// 调用 Map 的 clear 方法清空所有 Entry, 也就清空了 HashSet 中所有元素
public void clear()
{
    map.clear();
}
}
```

从上面的源程序可以看出, HashSet 的实现其实非常简单, 它只是封装了一个 HashMap 对象来存储所有的集合元素, 所有放入 HashSet 中的集合元素实际上由 HashMap 的 key 来保存, 而 HashMap 的 value 则存储了一个 PRESENT, 它是一个静态的 Object 对象。HashSet 的绝大部分方法都是通过调用 HashMap 的方法来实现的, 因此 HashSet 和 HashMap 两个集合在实现本质上是相同的。

我们来看一个例子, 代码清单 3-54 演示了 HashSet 的应用。

代码清单 3-54 HashSet 应用

```
import java.util.HashSet;
import java.util.Set;

public class testHashTable {
    public static void main(String[] args)
    {
        Set<Name> s = new HashSet<Name>();
        s.add(new Name("mingyao", "zhou"));
        System.out.println(s.contains(new Name("mingyao", "zhou")));
    }
}

class Name
{
    private String first;
    private String last;

    public Name(String first, String last)
    {
        this.first = first;
        this.last = last;
    }
}
```

```

public boolean equals(Object o)
{
    if (this == o)
    {
        return true;
    }

    if (o.getClass() == Name.class)
    {
        Name n = (Name)o;
        return n.first.equals(first)&& n.last.equals(last);
    }
    return false;
}
}

```

代码清单 3-54 的运行结果是 false。程序中向 HashSet 里添加了一个 new Name("mingyao", "zhou") 对象之后,立即通过程序判断该 HashSet 是否包含一个 new Name("mingyao", "zhou")对象。粗看上去,很容易以为该程序会输出 true。实际运行程序将看到程序输出 false,这是因为 HashSet 判断两个对象相等的标准除了要求通过 equals()方法比较返回 true 之外,还要求两个对象的 hashCode()返回值相等。而上面程序没有重写 Name 类的 hashCode()方法,两个 Name 对象的 hashCode()返回值并不相同,因此 HashSet 会把它们当成两个对象处理,因此程序返回 false。由此可见,当我们试图把某个类的对象当成 HashMap 的 key,或试图将这个类的对象放入 HashSet 中保存时,重写该类的 equals(Object obj)方法和 hashCode()方法很重要,而且这两个方法的返回值必须保持一致。当该类的两个 hashCode()返回值相同时,它们通过 equals()方法比较也应该返回 true。通常来说,所有参与计算 hashCode()返回值的关键属性,都应该用于作为 equals()比较的标准。

代码清单 3-55 重写了 equals()方法和 hashCode()方法,运行后输出是 true 了。

代码清单 3-55 HashSet 应用

```

import java.util.HashSet;
import java.util.Set;

public class testHashTable {
    public static void main(String[] args)
    {
        Set<Name> s = new HashSet<Name>();
        s.add(new Name("mingyao", "zhou"));
        System.out.println(s.contains(new Name("mingyao", "zhou")));
    }
}

class Name
{
    private String first;
    private String last;
}

```



```

public Name(String first, String last)
{
    this.first = first;
    this.last = last;
}

// 根据 first 判断两个 Name 是否相等
public boolean equals(Object o)
{
    if (this == o)
    {
        return true;
    }
    if (o.getClass() == Name.class)
    {
        Name n = (Name)o;
        return n.first.equals(first);
    }
    return false;
}

// 根据 first 计算 Name 对象的 hashCode() 返回值
public int hashCode()
{
    return first.hashCode();
}

public String toString()
{
    return "Name[first=" + first + ", last=" + last + "];"
}
}

```

上面程序中提供了一个 Name 类，该 Name 类重写了 equals()、hashCode()、toString()三个方法，这两个方法都是根据 Name 类的 first 实例变量来判断的，当两个 Name 对象的 first 实例变量相等时，这两个 Name 对象的 hashCode()返回值也相同，通过 equals()比较也会返回 true。程序 main 函数先将第一个 Name 对象添加到 HashSet 中，该 Name 对象的 first 实例变量值为"abc"，接着程序再次试图将一个 first 为"abc"的 Name 对象添加到 HashSet 中，很明显，此时没法将新的 Name 对象添加到该 HashSet 中，因为此处试图添加的 Name 对象的 first 也是"abc"，HashSet 会判断此处新增的 Name 对象与原有的 Name 对象相同，因此无法添加进入，程序输出 set 集合时将看到该集合里只包含一个 Name 对象，就是第一个、last 为"123"的 Name 对象。

3.2.8 LinkedHashMap、TreeMap 比较

LinkedHashMap 继承自 HashMap，同时在 HashMap 的基础上又在内部增加了一个链表，用以存放元素的顺序，所以它的性能整体较快。

HashMap 通过 hash 算法可以最快速地进行 put()和 get()操作。TreeMap 则提供了一种完全不同的 Map 实现。从功能上讲，TreeMap 有着比 HashMap 更为强大的功能，它实现了 SortedMap 接口，

这意味着它可以对元素进行排序。`TreeMap` 的性能略微低于 `HashMap`。如果在开发中需要对元素进行排序,那么使用 `HashMap` 便无法实现这种功能,使用 `TreeMap` 的迭代输出将会以元素顺序进行。`LinkedHashMap` 是基于元素进入集合的顺序或者被访问的先后顺序排序,`TreeMap` 则是基于元素的固有顺序(由 `Comparator` 或者 `Comparable` 确定)。

`LinkedHashMap` 是根据元素增加或者访问的先后顺序进行排序,而 `TreeMap` 则根据元素的 `key` 进行排序。

代码清单 3-56 所示实例演示了使用 `TreeMap` 实现业务逻辑的排序。

代码清单 3-56 `TreeMap` 应用

```
import java.util.Iterator;
import java.util.Map;
import java.util.TreeMap;

public class Student implements Comparable<Student>{

    public String name;
    public int score;
    public Student(String name,int score){
        this.name = name;
        this.score = score;
    }

    @Override
    //告诉 TreeMap 如何排序
    public int compareTo(Student o) {
        // TODO Auto-generated method stub
        if(o.score<this.score){
            return 1;
        }else if(o.score>this.score){
            return -1;
        }
        return 0;
    }

    @Override
    public String toString(){
        StringBuffer sb = new StringBuffer();
        sb.append("name:");
        sb.append(name);
        sb.append(" ");
        sb.append("score:");
        sb.append(score);
        return sb.toString();
    }

    public static void main(String[] args){
```



```
TreeMap map = new TreeMap();
Student s1 = new Student("1",100);
Student s2 = new Student("2",99);
Student s3 = new Student("3",97);
Student s4 = new Student("4",91);
map.put(s1, new StudentDetailInfo(s1));
map.put(s2, new StudentDetailInfo(s2));
map.put(s3, new StudentDetailInfo(s3));
map.put(s4, new StudentDetailInfo(s4));

//打印分数位于 S4 和 S2 之间的人
Map map1=((TreeMap)map).subMap(s4, s2);
for(Iterator iterator=map1.keySet().iterator();iterator.hasNext();){
    Student key = (Student)iterator.next();
    System.out.println(key+"->"+map.get(key));
}
System.out.println("subMap end");

//打印分数比 s1 低的人
map1=((TreeMap)map).headMap(s1);
for(Iterator iterator=map1.keySet().iterator();iterator.hasNext();){
    Student key = (Student)iterator.next();
    System.out.println(key+"->"+map.get(key));
}
System.out.println("subMap end");

//打印分数比 s1 高的人
map1=((TreeMap)map).tailMap(s1);
for(Iterator iterator=map1.keySet().iterator();iterator.hasNext();){
    Student key = (Student)iterator.next();
    System.out.println(key+"->"+map.get(key));
}
System.out.println("subMap end");
}

}

class StudentDetailInfo{
    Student s;
    public StudentDetailInfo(Student s){
        this.s = s;
    }
    @Override
    public String toString(){
        return s.name + "'s detail information";
    }
}
```

代码 3-56 程序运行后的输出如代码清单 3-57 所示。

代码清单 3-57 3-56 代码运行输出

```
name:4 score:91->4's detail information
name:3 score:97->3's detail information
subMap end
name:4 score:91->4's detail information
name:3 score:97->3's detail information
name:2 score:99->2's detail information
subMap end
name:1 score:100->1's detail information
subMap end
```

3.2.9 集合处理优化新方案

编程语言一般都需要提供一种机制遍历软件对象的集合，现代的编程语言支持更为复杂的数据结构，如列表、集合、映射和数。遍历能力是通过公共方法提供，而内部细节都隐藏在类的私有部分，所以程序员不需要了解其内部实现就能够遍历这些数据结构中的元素，这就是迭代的目的。迭代器是对集合中的所有元素进行顺序访问并可以对每个元素执行某些操作的机制。迭代器在本质上提供了在封装的对象集合上做“循环”的装置。

常见的使用迭代器的例子有：

- (1) 访问目录中的每个文件并显示文件名。
- (2) 访问队列中的每个客户（如银行排队）并判断用户等待了多久。使用迭代器时，一般情况下可以循环嵌套，即可以在同一时间做多个遍历。
- (3) 迭代器应该是无损的，即迭代行为不应该改变集合本身，如迭代时不要从集合中移除或插入元素。
- (4) 在某些情况下，你需要使用迭代器的不同遍历方法，例如，树的前序遍历和后序遍历，或者深度优先、广度优先遍历。

迭代器模式

迭代器设计模式是一种行为模式，其核心思想是负责访问和遍历列表中的对象，并把这些对象放到一个迭代器对象中。迭代器的实现方法根据谁来控制迭代分为两种：主动迭代和被动迭代。主动迭代器是由客户程序创建迭代器，调用 `next()` 行进到下一个元素，测试查看是否所有元素已被访问。被动迭代器是 Java8 新引入的机制，它是迭代器本身控制迭代，即迭代器自行 `next()` 向下走，针对客户程序来说迭代是透明的，是不能操作的。这种方法在 LISP 语言中很常见。

GOF⁵给出的定义是，在不暴露该对象的内部细节前提下，通过提供一种方法用于访问一个容器（container）对象中各个元素。深层次的目的是为了把遍历算法从容器对象中独立出来，算法如

⁵ 《Design Patterns: Elements of Reusable Object-Oriented Software》，由 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 合著（Addison-Wesley, 1995）。这几位作者常被称为“四人组（Gang of Four）”。

果和应用高度耦合，那最终会完全曲解了算法的发展方向，最终导致算法和数据挖掘⁶高度重复。

面向对象设计的一大难点是如何正确辨认对象的职责。理想状态下，一个类应该只有一个单一的职责⁷。职责分离可以最大限度地去除对象之间的耦合程度，但是实际开发过程中，想要做到职责单一着实不易。具体到本模式，以迭代器模式为例，容器对象提供了两个职责，一是组织管理数据对象，二是提供遍历算法。所以 Iterator 模式就是分离了集合对象的遍历行为，抽象出一个迭代器类来负责，这样既可以做到不暴露集合的内部结构，又可让外部代码透明地访问集合内部的数据。

迭代器模式由以下几个角色组成：

- 迭代器角色 (Iterator)：迭代器角色负责定义访问和遍历元素的接口。
- 具体迭代器角色 (Concrete Iterator)：具体迭代器角色要实现迭代器接口，并要记录遍历中的当前位置。
- 容器角色 (Container)：容器角色负责提供创建具体迭代器角色的接口。
- 具体容器角色 (Concrete Container)：具体容器角色实现创建具体迭代器角色的接口，这个具体迭代器角色与该容器的结构相关。

迭代器模式的类图如图 3-1 所示。

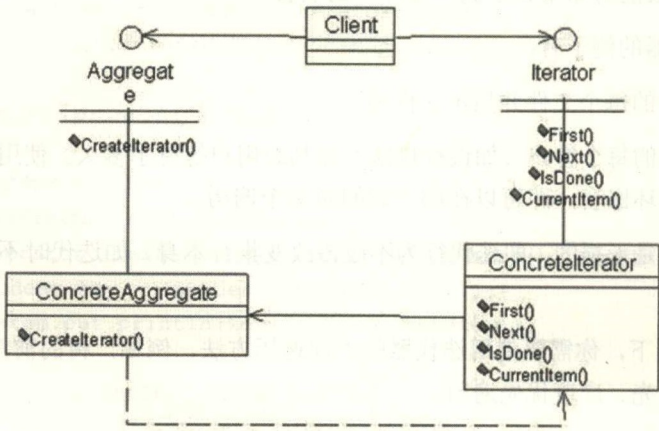


图3-1 迭代器模式类图

从上面的图可以看出，在 JDK 内部，与迭代器相关的接口有两个 Iterator、Iterable。Iterator 及其子类通常是迭代器本身的结构与方法。Iterable 是可迭代的，如 ArrayList HashMap 等需要使用迭代器功能的类，都需要实现该接口。下面介绍 Iterator 和 Iterable 两个接口的源码。

Iterator 如清单 3-58 所示。

⁶ 数据挖掘（英语：Data Mining），又译为资料探勘、数据采矿。它是数据库知识发现（英语：Knowledge-Discovery in Databases，简称：KDD）中的一个步骤。数据挖掘一般是指从大量的数据中通过算法搜索隐藏于其中信息的过程。数据挖掘通常与计算机科学有关，并通过统计、在线分析处理、情报检索、机器学习、专家系统（依靠过去的经验法则）和模式识别等诸多方法来实现上述目标。

⁷ 即设计模式中提到的单一职责。

代码清单 3-58 Iterator 源代码

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Iterable 如代码清单 3-59 所示。

代码清单 3-59 Iterable 源代码

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

实际应用过程中，我们如何来使用迭代器呢？

- (1) 比如类 A 想要使用迭代器，它的类声明部分应该是 `class A implement Iterable`;
- (2) 在类 A 实现中，要实现 Iterable 接口中的唯一方法：`Iterator<T> iterator()`;这个方法用于返回一个迭代器，即 Iterator 接口及其子类；
- (3) 在类 A 中，定义一个内部类 S，专门用于实现 Iterator 接口，定制类 A 自己的迭代器实现。

具体实现代码如下所清单 3-60 所示。

代码清单 3-60 定制迭代器类实现

```
class iteImplemtion implement Iterable
{
    Iterator<T> iterator() {...}
    class S implement Iterator<E>
    {
        boolean hasNext() {...}
        E next() {...}
        void remove() {...}
    }
}
```

Lambda 表达式

Java8 引入了独特的 Lambda 表达式用于遍历集合。Lambda 表达式本质上是一个匿名方法，我们来看下面这个例子。

代码清单 3-61 add 方法

```
public int add(int x, int y) {
    return x + y;
}
```

转成 Lambda 表达式后就变成`(int x, int y) -> x + y`;这一行表达式。参数类型可以省略，Java 编

译器会根据上下文推断出来:

```
(x, y) -> x + y; //返回两数之和
```

或者

```
(x, y) -> { return x + y; } //显式指明返回值
```

可见 Lambda 表达式由三部分组成: 参数列表, 箭头 (->), 以及一个表达式或语句块。

下面这个例子中的 Lambda 表达式没有参数, 也没有返回值, 即相当于一个方法接受 0 个参数, 返回 void, JDK 里 Runnable 接口的 run 方法就是这样一个实现。

```
() -> { System.out.println("Hello Lambda!"); }
```

如果只有一个参数且可以被 Java 推断出类型, 那么参数列表的括号也可以省略:

```
c -> { return c.size(); }
```

Java1.0 和 1.1 种两个主要的集合类是 Vector 和 Hashtable, 迭代器是通过一个叫作枚举的类实现的。今天无论是 Vector 还是 Hashtable 都是泛型类, 但退回到那时泛型还不是 Java 语言的一部分, 泛型是在 JDK5 的时候被引入的。清单 3-62 所示代码演示了使用枚举来处理字符串向量的方法。

代码清单 3-62 使用枚举处理字符串

```
Vector names = new Vector();
names.add("Apple");
names.add("Orange");
Enumeration e = names.elements();
while (e.hasMoreElements())
{
    String name = (String) e.nextElement();
    System.out.println(name);
}
```

Java8 提供了全新的遍历对象集合方式, 该方式主要包含主动迭代、流、并行流三种方法。总的来说, Java8 提供的迭代器较 JDK 早期版本而言, 它的可读性更好、不易出错、更容易并行化。进入 Java8 的新方式学习之前, 我们先来回顾一下集合类访问的变化过程。

Java1.2 推出了集合类 (Collections), 并通过一个迭代器类 (Iterator) 实现了迭代器设计模式。因为 Java1.2 当时还没有推出泛型概念, 所以我们需要对迭代器返回的对象进行强制类型转换。对于 Java1.2 至 1.4 版本, 遍历字符串列表方式如代码清单 3-63 所示。

代码清单 3-63 Java1.2 便利字符串列表方式

```
List names = new LinkedList();
names.add("Apple");
names.add("Orange");
Iterator i = names.iterator();
while (i.hasNext())
{
```

```
String name = (String) i.next();
System.out.println(name);
}
```

Java5 提出了泛型、Iterator 接口、增强 for 循环这三种新的方式。在增强 for 循环中，迭代器的创建和调用它的 hasNext() 和 next() 方法都发生在程序后端，不需要明确地写在代码中，因此，代码显得更为紧凑。Java5 的例子代码如清单 3-64 所示。

代码清单 3-64 Java5 处理方式

```
List<String> names = new LinkedList<String>();
names.add("Apple");
names.add("Orange");
for (String name : names)
System.out.println(name);
```

Java7 为了避免泛型的冗长给出了钻石运算符<>，从而避免了使用 new 运算符实例化泛型类时重复指定数据类型。从 Java7 开始，第一行代码可以简化成以下形式：

```
List<String> names = new LinkedList<>();
```

Java8 提供了新的迭代途径，它使用之前介绍的 lambda 表达式对集合进行遍历。Java8 最主要的新特性就是 lambda 表达式以及与此相关的特性，如流(streams)、方法引用(method references)、功能接口(functional interfaces)。正是因为这些新特性，我们能够使用被动迭代器而不是传统的主动迭代器，特别是 Iterable 接口提供了一个被动迭代器的默认方法叫作 forEach()。默认方法是 Java8 的又一个新特性，是一个接口方法的默认实现，在这种情况下，forEach() 方法实际上是用类似于 Java5 这样的主动迭代器方式来实现的。

实现了 Iterable 接口的集合类（如：所有列表 List、集合 set）现在都有一个 forEach() 方法，这个方法接收一个功能接口参数，实际上传递给 forEach() 方法的参数是一个 lambda 表达式。使用 Java8 的功能，代码变化如下所示，该代码易读性更好，且多线程环境下逻辑是线程安全的，更容易进行并行化。

代码清单 3-65 Java8 处理方式

```
List<String> names = new LinkedList<>();
names.add("Apple");
names.add("Orange");
names.forEach(name-> System.out.println(name));
```

请注意上述代码中的被动迭代与前面三段代码中的主动迭代之间的差异。在主动迭代中由循环结构控制迭代，并且每次通过循环从列表中获取一个对象，然后打印出来。上面的代码中没有显示的循环结构，我们只是告诉 forEach() 方法对列表中的对象实施打印，迭代控制隐含在 forEach() 方法中。

流是应用在一组元素上的一次执行的操作序列。集合和数组都可以用来产生流，因此称作数据流。流不存储集合中的元素，相反，流失通过管道操作来自数据源的值序列的一种机制。流管道由数据源、若干个中间操作(Intermediate Operations)、一个最终操作(Terminal Operation)组

成，中间操作对数据集完成过滤、检索等中间业务，而最终操作完成对数据集处理的最终处理，或调用 `forEach()` 方法。

当你处理集合时，通常会迭代所有元素并对其中的每一个进行处理。例如，假设我们希望统计一个文件中的所有长单词(超过 12 个字母以上的单词我们认为是长单词)。代码如清单 3-66 所示。

代码清单 3-66 Java7 统计长单词出现次数

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.Arrays;
import java.util.List;

public class Java7CollectionTest {
    public static void main(String[] args){
        try {
            String contents = new String(Files.readAllBytes(
                Paths.get("D:\\Project\\Java8Project\\src\\pom.xml")),
                StandardCharsets.UTF_8);
            List<String> words = Arrays.asList(contents.split("\n"));
            //进行迭代
            int count = 0;
            for(String w: words){
                if(w.length() > 12) count++;
            }
            System.out.println(count);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

这里面有什么错误吗？其实没有——只是它很难被并行计算。这也是 Java8 引入大量操作符的原因。在 Java8 中，实现相同功能的操作符如代码清单 3-67 所示。

代码清单 3-67 Java8 统计长单词出现次数

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Arrays;
import java.util.List;
```

```
public class Java8CollectionTest {
    public static void main(String[] args) {
        try {
```

```

String contents = new String(Files.readAllBytes(
Paths.get("/home/fuhd/work/workspace/javaee/wwos.platform/pom.xml")),
StandardCharsets.UTF_8);
List<String> words = Arrays.asList(contents.split("\n"));
//注意这一句
long count = words.stream().filter(w -> w.length() > 12).count();
System.out.println(count);
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

Stream 方法会为单词列表生成一个 **Stream**。**Filter** 方法会返回另一个只包含单词长度大于 12 的 **Stream**，**Count** 方法会将 **Stream** 化简为一个结果。

一个 **Stream** 表面上看与一个集合很类似，允许你改变和获取数据。但是实际上它与集合是有很大区别的：

- (1) **Stream** 自己不会存储元素。元素可能被存储在底层的集合中，或者根据需要产生出来；
- (2) **Stream** 操作符不会改变源对象。相反，它们会返回一个持有结果的新 **Stream**；
- (3) **Stream** 操作符可能是延迟执行的。这意味着它们会等到需要结果的时候才执行。

许多人发现 **Stream** 表达式比循环的可读性更好。此外，它们还很容易进行并行执行。代码清单 3-68 包含的是一段如何并行统计长单词的代码。

代码清单 3-68 Java8 并行统计单词

```

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Arrays;
import java.util.List;

public class T8 {
    public static void main(String[] args) {
        try {
            String contents = new String(Files.readAllBytes(
                Paths.get("/home/fuhd/work/workspace/javaee/wwos.platform/
pom.xml")),
                StandardCharsets.UTF_8);
            List<String> words = Arrays.asList(contents.split("\n"));
            //注意这一句，stream()改成了parallelStream()方法
            long count = words.parallelStream().filter(w -> w.length() > 12).count();
            System.out.println(count);
        } catch (IOException e) {

```



```

        e.printStackTrace();
    }
}
}

```

只要将 `stream()` 方法改成 `parallelStream` 方法, 就可以让 Stream API 并行执行过滤和统计操作。

Stream 遵循“做什么, 而不是怎么去做”的原则。在我们的示例中, 描述了需要做什么: 获得长单词并对它们的个数进行统计。我们没有指定按照什么顺序, 或者在哪个线程中做, 它们都是理所应当发生的。相反, 循环一开始就需要指定如何进行计算, 因此就失去了优化的机会。

当你使用 Stream 时, 你会通过三个阶段来建立一个操作流水线:

(1) 创建一个 Stream;

(2) 在一个或多个步骤中, 指定将初始 Stream 转换为另一个 Stream 的中间操作;

(3) 使用一个终止操作来产生一个结果。该操作会强制它之前的延迟操作立即执行。在这之后, 该 Stream 就不会再被使用。

在我们的示例中, 通过 Stream 或者 `parallelStream` 方法来创建 Stream, 再通过 `filter` 方法对其进行转换, 而 `count` 就是终止操作。

注意: Stream 操作不会按照元素的调用顺序执行。在我们的例子中, 只有在 `count` 被调用的时候才会执行 Stream 操作。当 `count` 方法需要第一个元素时, `filter` 方法会开始请求各个元素, 直到找到一个长度大于 12 的元素。

以下代码使用流管道计算以字母 A 开头的人名的个数, 是 Java8 代码。列表 `names` 用于创建流, 然后使用过滤器对数据集进行过滤, `filter()` 方法只过滤出以字母 A 开头的名字, 该方法的参数是一个 lambda 表达式。最后, 流的 `count()` 方法作为最终操作, 得到应用结果。中间操作除了 `filter()` 之外, 还有 `distinct()`、`sorted()`、`map()` 等, 一般是对数据集的整理, 返回值一般也是数据集。清单 3-69 代码使用的是主动式迭代方式, 在多线程环境下该逻辑是线程不完全的。

代码清单 3-69 主动式迭代方法

```

List<String> names = new LinkedList<>();
names.add("Annie");
names.add("Alice");
names.add("Bob");
long count = names.stream()
    .filter(name -> name.startsWith("A")).count();

```

同样的代码在 Java7 里如清单 3-70 所示。

代码清单 3-70 Java7 主动式迭代方法

```

List<String> names = new LinkedList<>();
names.add("Annie");
names.add("Alice");
names.add("Bob");
long count = 0;

```

```
for (String name : names){
    if (name.startsWith("A"))
        ++count;
}
```

最终方法往往是完成对数据集中数据的处理，如 `forEach()`，还有 `allMatch()`、`anyMatch()`、`findAny()`、`findFirst()`，数值计算类的方法有 `sum()`、`max()`、`min()`、`averag()`等，最终方法也可以是对集合的处理，如 `reduce()`、`collect()`等。`reduce()`方法的处理方式一般是每次都产生新的数据集，而 `collect()`方法是在原数据集的基础上进行更新，过程中不产生新的数据集。

无论是从易读写的角度来说，还是从 API 设计的角度来说迭代器、`Iterable` 接口和 `foreach` 循环都是非常好用的。但代价是，使用它们时是会额外在堆上为每个循环子创建一个对象。如果循环要执行很多很多遍，请注意避免生成无意义的实例，最好用基本的指针循环方式来代替上述迭代器、`Iterable` 接口和 `foreach` 循环。

Java8 集合类不仅具有 `Stream()`方法，该方法返回一个连续的数据流，Java8 集合类还有一个 `parallelStream()`方法，该方法返回一个并行流。并行流的作用在于允许管道操作同时在不同的 Java 线程中执行以提高性能。但要注意的是，集合元素的处理顺序可能发生改变。

Java8 支持一种新功能进行迭代，它是声明性的方法，这种新方法带来的好处是代码的可读性更好，不易出错，对于多线程支持更好、更丰富，程序更容易并行化。然而测试表明，并行流对每一种集合类而言不一定性能更快。

3.2.10 优先考虑并行计算

当在多核处理器上部署 Java 程序时，由于 Java 7 引入的 `ForkJoinPool` 和 Java 8 引入的并行数据流⁸都对并行处理有所帮助，所以在多核环境下表现尤为明显，这是因为所有的处理器都可以做到访问相同的内存。这两个技术我们会在第 5 章具体深入介绍，本小节只给出一些基本介绍。

数据流的特点包括如下几点。

- **元素序列**：流提供了一组特定类型的以顺序方式元素。流获取/计算需求的元素。它不存储元素。
- **源**：流使用集合，数组或 I/O 资源为输入源。
- **聚合操作**：数据流支持如 `filter`、`map`、`limit`、`reduced`、`find`、`match` 等聚合操作。
- **管道传输**：大多数流操作的返回流本身使他们的结果可以被管道传输。这些操作被称为中间操作以及它们的功能是利用输入，处理输入和输出返回到目标。`collect()`方法是终端操作，这是通常出现在管道传输操作结束标记流的结束。
- **自动迭代**：流操作内部做了反复对比，其中明确迭代需要集合提供源元素。

现在，类似 Scala 这样的函数式编程语言都鼓励使用递归。因为递归通常意味着能分解到单独个体优化的尾递归（tail-recursing）。如果你使用的编程语言能够支持那是再好不过的了，不过即使如此，也要注意对算法的细微调整会让尾递归变为普通递归。所以我们希望编译器能自动探测

⁸ 流代表从支持聚合操作源的序列的对象。

到这一点，否则本来我们将为只需使用几个本地变量就能搞定的事情而白白浪费大量的堆栈框架（stack frames）。

JDK7 新增了并发框架-fork/join 框架，在这种框架下，ForkJoinTask 代表一个需要执行的任务，真正执行这些任务的线程是放在一个线程池（ForkJoinPool）里面。ForkJoinPool 是一个可以执行 ForkJoinTask 的 ExcuteService，与 ExcuteService 不同的是它采用了 work-stealing 模式，即所有在池中的线程尝试去执行其他线程创建的子任务，这样就很少有线程处于空闲状态，非常高效。

总的来说，Java 8 引入批量操作接口的目的是为了给 Java 集合类库增加批量操作数据的支持。通常称这种批量数据操作为“Java 中的 filter/map/reduce”。批量数据操作有串行（在当前线程上）和并行（使用多线程）两种操作模式。Java 8 并行流（parallel stream）采用共享线程池，对性能造成了严重影响。可以包装流来调用自己的线程池解决性能问题。所以，这种并行处理较之在跨网络的不同机器上进行扩展，根本的好处是几乎可以完全消除延迟。

虽然并行计算的效果很明显，但是我们需要注意以下两点。

- （1）并行处理会吃光处理器资源。并行处理为批处理带来了极大的好处，但同时也是非同步服务器（如 HTTP）的噩梦。为什么在过去的几十年中我们一直在使用单线程的 Servlet 模型，这是因为并行处理仅在纵向扩展时才能带来实际的好处。
- （2）并行处理对算法复杂度没有影响。如果你的算法的时间复杂度为 $O(n\log n)$ ，让算法 X 个处理器上运行，事件复杂度仍然为 $O(n\log n/x)$ ，因为 X 只是算法中的一个无关紧要的常量。你节省的仅仅是时钟时间（wall-clock time），实际的算法复杂度并没有降低。

降低算法复杂度毫无疑问是改善性能最行之有效的办法。比如对于一个 HashMap 实例的 lookup()方法来说，事件复杂度 $O(1)$ 或者空间复杂度 $O(1)$ 是最快的。如果你不能降低算法的复杂度，也可以通过找到算法中的关键点并加以改善的方法，来起到改善性能的作用。

3.3 字符串概念

3.3.1 String 对象

String 对象是我们日常使用的对象类型，字符串对象或者其等价对象(如 char 数组)，在内存中总是占据了最大的空间块，因此如何高效地处理字符串，是提高系统整体性能的关键。

一个 Java 对象实际还会占用些额外的空间，如：对象的 class 信息、ID、在虚拟机中的状态。在 Oracle JDK 的 Hotspot 虚拟机中，一个普通的对象需要额外 8 个字节。

如果对于 String（JDK6）的成员变量声明如代码清单 3-71 所示。

代码清单 3-71 String 成员变量声明

```
private final char value[];  
private final int offset;  
private final int count;  
private int hash;
```

那么应该如何计算该 String 所占的空间？

首先计算一个空的 char 数组所占空间,在 Java 里数组也是对象,因而数组也有对象头,所以一个数组所占的空间为对象头所占的空间加上数组长度,即 $8+4=12$ 个字节,经过填充后为 16 个字节。那么一个空 String 所占空间为:对象头(8 字节)+char 数组(16 字节)+3 个 int ($3\times 4=12$ 字节)+1 个 char 数组的引用(4 字节)=40 字节。因此一个实际的 String 所占空间的计算公式就是, $8*((8+2*n+4+12)+7)/8=8*(int)((n*2)+43)/8$ 其中, n 为字符串长度。

String 对象可以认为是 char 数组的延伸和进一步封装,它主要由 3 部分组成:char 数组、偏移量和 String 的长度。char 数组表示 String 的内容,它是 String 对象所表示字符串的超集。String 的真实内容还需要由偏移量和长度在这个 char 数组中进行定位和截取。

String 对象的创建方式很有讲究,关键是要明白它内部的实现原理。以下列举了 4 点原理。

- (1) 当使用任何方式来创建一个字符串对象 X 时,Java 运行时(运行中 JVM)会拿着这个 X 在 String 池中查找是否存在内容相同的字符串对象,如果不存在,则在池中创建一个字符串 X,否则,不会创建对象,即不会在池中添加;
- (2) 前面说过,Java 内部只要使用 new 关键字来创建对象,则一定会(在堆区或栈区)创建一个新的对象;
- (3) 使用直接指定或者使用纯字符串串联来创建 String 对象,则仅仅会检查维护 String 池中的字符串,池中如果没有就创建一个,如果存在,就不需要创建新的,但绝不会在堆栈区再去创建该 String 对象;
- (4) 使用包含变量的表达式来创建 String 对象,则不仅会检查并维护 String 池,而且还会在堆栈区创建一个 String 对象。

在拼接静态字符串时,尽量用+,因为通常编译器会对此做优化,如 `String test = "this" + "is" + "a" + "test" + "string"`,编译器会把它视为 `String test = "this is a test string"`。所以在拼接动态字符串时,尽量用 `StringBuffer` 或 `StringBuilder` 的 `append`,这样可以减少构造过多的临时 String 对象。

String 有 3 个基本特点,即不变性、针对常量池的优化及类的 final 定义。

不变性指的是 String 对象一旦生成,则不能再对它进行改变。String 的这个特性可以泛化成不变(immutable)模式,即一个对象的状态在对象被创建之后就不再发生变化。不变模式的主要作用在于当一个对象需要被多线程共享,并且访问频繁时,可以省略同步和锁等待的时间,从而大幅提高系统性能。

针对常量池的优化指的是当两个 String 对象拥有相同的值时,它们只引用常量池中的同一个拷贝,当同一个字符串反复出现时,这个技术可以大幅度节省内存空间。

下面代码 `str1`、`str2`、`str4` 引用了相同的地址,但是 `str3` 却重新开辟了一块内存空间,虽然 `str3` 单独占用了堆空间,但是它所指向的实体和 `str1` 完全一样。代码清单 3-72 代码演示了 4 个 String 对象的引用方式对比。

代码清单 3-72 String 对象引用方式对比实验

```
public class StringDemo {
    public static void main(String[] args){
        String str1 = "abc";
```



```

String str2 = "abc";
String str3 = new String("abc");
String str4 = str1;
System.out.println("is str1 = str2?" + (str1==str2)); //检查是否 str1 和 str2
对象引用了相同内存地址
System.out.println("is str1 = str3?" + (str1==str3)); //检查是否 str1 和 str3
对象引用了相同内存地址
System.out.println("is str1 refer to str3?" + (str1.intern()==str3.intern()));
//检查是否 str1 的内容和 str3 相同
System.out.println("is str1 = str4?" + (str1==str4)); //检查是否 str1 和 str4 对
象引用了相同内存地址
System.out.println("is str2 = str4?" + (str2==str4)); //检查是否 str2 和 str4
对象引用了相同内存地址
System.out.println("is str4 refer to str3?" + (str4.intern()==str3.intern()));
//检查是否 str3 的内容和 str4 相同
    }
}

```

代码 3-72 运行输出如清单 3-73 所示。

代码清单 3-73 String 对象引用方式对比实验运行输出

```

is str1 = str2?true
is str1 = str3?false
is str1 refer to str3?true
is str1 = str4true
is str2 = str4true
is str4 refer to str3?true

```

程序代码 3-72 里面使用了 String 对象的 `intern()` 方法，`intern()` 方法是一个本地方法，定义为 `public native String intern();` `intern()` 方法的价值在于让开发者能将注意力集中到 String 池上。当调用 `intern` 方法时，如果池已经包含一个等于此 String 对象的字符串（该对象由 `equals(Object)` 方法确定），则返回池中的字符串。否则，将此 String 对象添加到池中，并且返回此 String 对象的引用。

当我们下意识地使用 `newString` 去构造一个全新的字符串而不是用赋值符来创建（重用）一个字符串时，就导致了另一个潜在的性能问题，即：重复创建大量相同的字符串。说到这里，您也许会想到使用缓存池的技术来解决这一问题，大概有如下两种方法。

- (1) 使用 String 的 `intern()` 方法返回 JVM 对字符串缓存池里相应已存在的字符串引用，从而解决内存性能问题。不建议采用这种方案的原因在于，首先，`intern()` 所使用的池会是 JVM 中一个全局的池，很多情况下我们的程序并不需要如此大作用域的缓存；其次，`intern()` 所使用的是 JVM heap 中 PermGen 相应的区域，在 JVM 中 PermGen 是用来存放装载类和创建类实例时用到的元数据。程序运行时所使用的内存绝大部分存放在 JVM heap 的其他区域，此外，过多地使用 `intern()` 将导致 PermGen 过度增长而最后返回 `OutOfMemoryError`，因为垃圾收集器不会对被缓存的 String 做垃圾回收。所以我们建议使用第二种方式。

- (2) 用户自己构建缓存，这种方式的优点是更加灵活。创建 `HashMap`，将需缓存的 String 作

为 key 和 value 存放入 HashMap。假设我们准备创建的字符串为 key，将 Map cacheMap 作为缓冲池，那么返回 key 的代码如清单 3-74 所示。

代码清单 3-74 Map cacheMap 缓存池

```
private String getCacheWord(String key) {
    String tmp = cacheMap.get(key);
    if(tmp != null) {
        return tmp;
    } else {
        cacheMap.put(key, key);
        return key;
    }
}
```

3.3.2 善用 String 对象的 SubString 方法

JDK6 中 String.substring(int, int)的源码为如清单 3-75 所示。

代码清单 3-75 subString 源代码

```
public String substring(int beginIndex, int endIndex) {
    if (beginIndex < 0) {
        throw new StringIndexOutOfBoundsException(beginIndex);
    }
    if (endIndex > count) {
        throw new StringIndexOutOfBoundsException(endIndex);
    }
    if (beginIndex > endIndex) {
        throw new StringIndexOutOfBoundsException(endIndex - beginIndex);
    }
    return ((beginIndex == 0) && (endIndex == count)) ? this :
        new String(offset + beginIndex, endIndex - beginIndex, value);
}
```

从上面的源代码可以看到，SubString 源代码里面有这么一行：“new String(offset+beginIndex, endIndex-beginIndex,value);” 该行代码的目的是为了能高效且快速地共享 String 内的 char 数组对象。

调用的 String 构造函数源码为如清单 3-76 所示。

代码清单 3-76 调用 String 构造函数源代码

```
String(int offset, int count, char value[]) {
    this.value = value;
    this.offset = offset;
    this.count = count;
}
```

在这种通过偏移量来截取字符串的方法中，String 的原生内容 value 数组被复制到新的子字符

串中。因此 `String.substring()` 所返回的 `String` 仍然会保存原始 `String`，设想，如果原始字符串很大，截取的字符长度却很短，那么截取的字字符串中包含了原生字符串的所有内容，并占据了相应的内存空间，而仅仅通过偏移量和长度来决定自己的实际取值。这就是为什么几万个平均长度的单词竟然占用了上百兆的内存的原因，显然这种算法提高了速度却浪费了空间，对于通过 `String.split()` 或 `String.substring()` 截取出大量 `String` 的操作，这种设计在很多时候可以很大程度的节省内存，因为这些 `String` 都复用了原始 `String`，只是通过 `int` 类型的 `start`，`end` 等值来标识每一个 `String`。而对于从一个巨大的 `String` 截取少数 `String` 为以后所用，这样的设计则造成大量冗余数据。

导致大量内存占用的根源是 `String.substring()` 返回结果中包含大量原始 `String`，那么减少内存浪费的途径就是去除这些原始 `String`。我们这里采取再次调用 `newString` 构造一个仅包含截取出的字符串的 `String`，我们可调用 `String.toCharArray()` 方法，`String newString = new String(smallString.toCharArray());` 举一个极端例子，假设要从一个字符串中获取所有连续的非空子串，字符串长度为 n ，如果用 JDK 本身提供的 `String.substring()` 方法，则总共的连续非空子串个数为 $n+(n-1)+(n-2)+\dots+1=n*(n+1)/2=O(n^2)$ ，由于每个子串所占的空间为常数，故空间复杂度也为 $O(n^2)$ 。

如果用本文建议的方法，即构造一个内容相同的新的字符串，则所需空间正比于子串的长度，则所需空间复杂度为 $1*n+2*(n-1)+3*(n-2)+\dots+n*1=(n^3+3*n^2+2*n)/6=O(n^3)$ 。所以，从以上定量的分析看来，当需要截取的字符串长度总和大于等于原始文本长度，我们这里建议的方法带来的空间复杂度反而高了，而现有的 `String.substring()` 设计恰好可以共享原始文本从而达到节省内存的目的。反之，当所需要截取的字符串长度总和远小于原始文本长度时，用我们推荐的方法会在很大程度上节省内存，在大文本数据处理中其优势显而易见。

代码清单 3-77 所示代码演示了使用 `substring` 方法在一个很大的 `string` 独享里面截取一段很小的字符串，如果采用 `string` 的 `substring` 方法会造成内存溢出，如果采用反复创建新的 `string` 方法可以确保正常运行。

代码清单 3-77 `String` 的 `subString` 示例

```
import java.util.ArrayList;

import java.util.List;

public class StringDemo {
    public static void main(String[] args){
        List<String> handler = new ArrayList<String>();
        for(int i=0;i<1000;i++){
            HugeStr h = new HugeStr();
            ImprovedHugeStr h1 = new ImprovedHugeStr();
            handler.add(h.getSubString(1, 5)); //加入集合
            handler.add(h1.getSubString(1, 5)); //加入集合
        }
    }
    //定义一个静态类 HugeStr
    static class HugeStr{
        private String str = new String(new char[800000]);
```

```

    public String getSubString(int begin,int end){
        return str.substring(begin, end);//调用 String 的 substring 方法
    }
}
//定义一个静态类 ImprovedHugeStr
static class ImprovedHugeStr{
    private String str = new String(new char[10000000]);
    public String getSubString(int begin,int end){
        return new String(str.substring(begin, end));//重新创建一个 String 对象
    }
}
}

```

上面的代码 3-77 运行后输出如 3-78 所示。

代码清单 3-78 String 的 substring 示例运行输出

```

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.Arrays.copyOf(Unknown Source)
    at java.lang.StringValue.from(Unknown Source)
    at java.lang.String.<init>(Unknown Source)
    at StringDemo$ImprovedHugeStr.<init>(StringDemo.java:23)
    at StringDemo.main(StringDemo.java:9)

```

上面的例子里面, ImprovedHugeStr 可以工作是因为它使用没有内存泄漏的 String 构造函数重新生成了 String 对象,使得由 substring()方法返回的,存在内存泄漏问题的 String 对象失去所有的强引用,从而被垃圾回收器识别为垃圾对象进行回收,保证了系统内存的稳定。

此外,很多文章建议读者采用 StringBuffer 替换 String 对象,但是有一种情况例外,即如果读者确定自己定义的字符串是常量字符串,那么建议不要使用 StringBuffer 类型定义这个常量字符串,还是采用 String 方式,因为常量字符串并不需要动态改变长度。我们使用 String 对象的 indexOf()和 substring()来分析字符串容易导致 stringindexoutofboundsexception 错误,而使用 stringtokenizer 类来分析字符串则会容易一些,效率也会高一些,也不会出现 stringindexoutofboundsexception 错误。

3.3.3 用 charat()代替 startswith()

如果只是查找单个字符的话,可以考虑用 charat()代替 startswith(),虽然用一个字符作为参数调用 startswith()从代码角度来看也不会抛错,但从性能角度上来看,这里还是采用 charat()方式比较快。

代码清单 3-79 startswith()方法示例

```

private void method(string s) {
    if (s.startswith("a")) {
        // ...
    }
}

```

我们可以将'startswith()'替换成'charat()',如代码清单 3-80 所示。

代码清单 3-80 charat()方法示例

```
private void method(string s) {  
    if ('a' == s.charAt(0)) {  
        // ...  
    }  
}
```

3.3.4 在字符串相加的时候, 使用'代替"

如果该字符串只有一个字符的话, 我们建议使用单引号, 而不是双引号。原始代码如清单 3-81 所示。

代码清单 3-81 字符串相加实验

```
public class strClass {  
    public void method(string s) {  
        string string = s + "d" // violation.  
        string = "abc" + "d" // violation.  
    }  
}
```

我们把上面程序的一个字符的字符串替换成单引号, 如代码清单 3-82 所示。

代码清单 3-82 字符串相加实验 strClass

```
public class strClass {  
    public void method(string s) {  
        string string = s + 'd'  
        string = "abc" + 'd'  
    }  
}
```

3.3.5 字符串切割

String API 支持传入正则表达式帮助处理字符串, 但是使用 Split 方式用作简单的字符串分割时性能较差。我们对比 String 对象的 split 方法和 StringTokenizer 类的处理字符串性能, 代码如清单 3-83 所示。

代码清单 3-83 处理字符串性能实验

```
import java.util.StringTokenizer;  
  
public class splitandstringtokenizer{  
    public static void main(String[] args){  
        String orgStr = null;  
        StringBuffer sb = new StringBuffer();  
        for(int i=0;i<100000;i++){  
            sb.append(i);  
            sb.append(",");  
        }  
    }  
}
```

```

    }
    orgStr = sb.toString();
    long start = System.currentTimeMillis();
    for(int i=0;i<100000;i++){
        orgStr.split(",");//调用 split 方法切割字符串
    }
    long end = System.currentTimeMillis();
    System.out.println(end-start);

    start = System.currentTimeMillis();
    String orgStr1 = sb.toString();
    StringTokenizer st = new StringTokenizer(orgStr1,",");
    for(int i=0;i<100000;i++){
        st.nextToken();
    }
    st = new StringTokenizer(orgStr1,","); //StringTokenizer 类切割字符串
    end = System.currentTimeMillis();
    System.out.println(end-start);

    start = System.currentTimeMillis();
    String orgStr2 = sb.toString();
    String temp = orgStr2;
    while(true){
        String splitStr = null;
        int j=temp.indexOf(",");
        if(j<0)break;
        splitStr=temp.substring(0, j);
        temp = temp.substring(j+1);//采用 String 对象的 subString 方法切割字符串
    }
    temp=orgStr2;
    end = System.currentTimeMillis();
    System.out.println(end-start);
}
}

```

代码清单 3-84 处理字符串性能实验运行输出, 单位毫秒

```

39015
16
15

```

上面的实验里面, 当一个 `StringTokenizer` 对象生成后, 通过它的 `nextToken()` 方法便可以得到下一个分割的字符串, 通过 `hasMoreToken` 方法可以知道是否有更多的字符串需要处理。对比发现 `split` 的耗时非常的长, 采用 `StringTokenizer` 对象处理速度很快。我们尝试自己实现字符串分割算法, 使用 `subString` 方法和 `indexOf` 方法组合而成的字符串分割算法可以帮助很快切分字符串并替换内容。

我们增加一个示例代码, 如果需要一个较大的字符串里面分割出数据, 效率差距很大。

代码清单 3-85 分割字符串性能实验

```
import java.util.StringTokenizer;

public class testSplit {
    public static void splitBefore(String startDate){
        long start = System.currentTimeMillis();
        StringBuffer stb = new StringBuffer();
        if(startDate!=null){
            //获取日期 yy-mm-dd
            int count = 0;
            for(int i=0;i<1000;i++){
                stb.append(startDate.split(" ")[i]);
                count++;
            }
            System.out.println(stb);
        }
        System.out.println(System.currentTimeMillis() - start);
    }

    public static void splitAfter(String startDate){
        long start = System.currentTimeMillis();
        StringBuffer stb = new StringBuffer();
        if(startDate!=null){
            //获取日期 yy-mm-dd
            int count = 0;
            StringTokenizer st = new StringTokenizer(startDate, " ");
            while(st.hasMoreElements()){
                stb.append(st.nextToken());
                count++;
            }
            System.out.println(stb);
        }
        System.out.println(System.currentTimeMillis() - start);
    }

    public static void main(String[] args){
        StringBuffer sb = new StringBuffer();
        for(int i=0;i<1000;i++){
            sb.append(i);
            sb.append(" ");
        }
        testSplit.splitBefore(sb.toString());
        testSplit.splitAfter(sb.toString());
    }
}
```

上面的运行输出，Split 方式相对来说多花费了 343 毫秒。可以看出，Split 方法确实比 StringTokenizer 类要慢很多。

`StringTokenizer` 类允许一个应用程序进入一个令牌 (token), `StringTokenizer` 类的对象在内部已经标识化的字符串中维持了当前位置。一些操作使得在现有位置上的字符串提前得到处理, 一个令牌的值是由获得其曾经创建 `StringTokenizer` 类对象的字符串所返回的。

除了上面介绍的, 使用 `stringtokenizer` 类来分析字符串也会容易一些, 效率也相对于 `indexOf()` 和 `substring()` 这两个方法来说要高一些, 所以对于需要应用 `indexOf()` 和 `substring()` 这两个方法的应用场景, 建议使用 `stringtokenizer` 代替。

3.3.6 字符串重编码

计算机中储存的信息都是用二进制数表示的; 而我们在屏幕上看到的英文、汉字等字符是二进制数转换之后的结果。通俗的说, 按照何种规则将字符存储在计算机中, 如 ‘a’ 用什么表示, 称为“编码”; 反之, 将存储在计算机中的二进制数解析显示出来, 称为“解码”, 如同密码学中的加密和解密。在解码过程中, 如果使用了错误的解码规则, 则导致 ‘a’ 解析成 ‘b’ 或者乱码。

字符集 (Charset): 是一个系统支持的所有抽象字符的集合。字符是各种文字和符号的总称, 包括各国家文字、标点符号、图形符号、数字等。

字符编码 (Character Encoding): 是一套法则, 使用该法则能够对自然语言的字符的一个集合 (如字母表或音节表), 与其他东西的一个集合 (如号码或电脉冲) 进行配对。即在符号集合与数字系统之间建立对应关系, 它是信息处理的一项基本技术。通常人们用符号集合 (一般情况下就是文字) 来表达信息。而以计算机为基础的信息处理系统则是利用元件 (硬件) 不同状态的组合来存储和处理信息的。元件不同状态的组合能代表数字系统的数字, 因此字符编码就是将符号转换为计算机可以接受的数字系统的数, 称为数字代码。

常见字符集名称: ASCII 字符集、GB2312 字符集、BIG5 字符集、GB18030 字符集、Unicode 字符集等。计算机要准确的处理各种字符集文字, 需要进行字符编码, 以便计算机能够识别和存储各种文字。

代码清单 3-86 所示的范例说明了怎么可以正确地对字符集进行转码。代码清单 3-87 是程序运行的输出。

代码清单 3-86 字符串重编码实验

```
import java.io.UnsupportedEncodingException;

/**
 * 字符串转码测试
 *
 */
public class TestEncoding {
    //如果出现缺少字符集的情况, 抛出字符编码异常
    //UTF-8 (8-bit Unicode Transformation Format) 是一种针对 Unicode 的可变长度字符编码 (定长码), 也是一种前缀码。它可以用来表示 Unicode 标准中的任何字符, 且其编码中的第一个字节仍与 ASCII 兼容, 这使得原来处理 ASCII 字符的软件无须或只需做少部分修改, 即可继续使用。因此, 它逐渐成为电子邮件、网页及其他存储或传送文字的应用中, 优先采用的编码。互联网工程工作小组 (IETF) 要求所有互联网协议都必须支持 UTF-8 编码。
```



```

        public static void main(String[] args) throws UnsupportedEncodingException {
            System.out.println("转码前, 输出 Java 系统属性如下: ");
            System.out.println("user.country:" + System.getProperty("user.country"));
//打印国家
            System.out.println("user.language:" + System.getProperty("user.language"));
//打印语言
            System.out.println("sun.jnu.encoding:" + System.getProperty("sun.jnu.encoding"));
//打印字符集
            System.out.println("file.encoding:" + System.getProperty("file.encoding"));
//打印字符集

            System.out.println("转码之后的输出: ");
            String s = "麦克周";
            String s1 = new String(s.getBytes(), "UTF-8");
            String s2 = new String(s.getBytes("UTF-8"), "UTF-8"); //如果你转码之后,
            又用该码来构建一个字符串, 是绝对不会出现乱码的
            String s3 = new String(s.getBytes("UTF-8"));
            String s4 = new String(s.getBytes("UTF-8"), "GBK");
            String s5 = new String(s.getBytes("GBK"));
            String s6 = new String(s.getBytes("GBK"), "GBK");
            System.out.println(s1); //输出乱码
            System.out.println(s2); //输出正确的中文
            System.out.println(s3); //输出乱码
            System.out.println(s4); //输出乱码
            System.out.println(s5); //输出正确的中文
            System.out.println(s6); //输出正确的中文
        }
    }
}

```

代码清单 3-87 字符串重编码实验运行输出

转码前, 输出 Java 系统属性如下:

转码前, 输出 Java 系统属性如下:

user.country:CN

user.language:zh

sun.jnu.encoding:GBK

file.encoding:GBK

转码之后的输出:

?????

麦克周

楡〰厠鍛?

楡〰厠鍛?

麦克周

麦克周

3.3.7 合并字符串

由于 String 是不可变对象, 因此, 在需要对字符串进行修改操作时(如字符串连接、替换), String 对象会生成新的对象, 所以其性能相对较差。但是 JVM 会对代码进行彻底的优化, 将多个连接操

作的字符串在编译时合成一个单独的长字符串。

针对超大的 String 对象，我们采用 String 对象连接、使用 concat 方法连接、使用 StringBuilder 类等三种方式，代码如清单 3-88 所示。

代码清单 3-88 合并字符串实验

```
public class StringConcat {
    public static void main(String[] args){
        String str = null;
        String result = "";

        long start = System.currentTimeMillis();
        for(int i=0;i<10000;i++){
            str = str + i;
        }
        long end = System.currentTimeMillis();
        System.out.println(end-start);

        start = System.currentTimeMillis();
        for(int i=0;i<10000;i++){
            result = result.concat(String.valueOf(i));
        }
        end = System.currentTimeMillis();
        System.out.println(end-start);

        start = System.currentTimeMillis();
        StringBuilder sb = new StringBuilder();
        for(int i=0;i<10000;i++){
            sb.append(i);
        }
        end = System.currentTimeMillis();
        System.out.println(end-start);
    }
}
```

代码清单 3-89 合并字符串实验运行输出

```
375
187
0
```

虽然第一种方法时编译器判断 String 的激发运行成 StringBuilder 实现，但是编译器没有做出足够聪明的判断，每次循环都生成了新的 StringBuilder 实例从而大大降低了系统性能。

StringBuffer 和 StringBuilder 都实现了 AbstractStringBuilder 抽象类，拥有几乎相同的对外借口，两者的最大不同在于 StringBuffer 对几乎所有的方法都做了同步，而 StringBuilder 并没有任何同步。由于方法同步需要消耗一定的系统资源，因此，StringBuilder 的效率也好于 StringBuffer。但是，在多线程系统中，StringBuilder 无法保证线程安全，不能使用。无论 StringBuilder 还是 StringBuffer，

底层使用的数据存储都是 `char[]`。随着新元素不断加入 `StringBuffer` 或 `StringBuilder`，底层的数据存储 `char[]` 的大小也需要随之调整。大小调整的结果是字符数组 `char[]` 分配到了一块更大空间创建新的字符数组 `char[]`，老数组中的字符元素被复制到了新的数组之中，而原有的老数组则被丢弃，即这部分资源会进行垃圾收集。底层使用数组存储数据的 Java Collection 类，也采用类似的方法管理内存。

代码清单 3-90 StringBuffer 和 StringBuilder 对比实验

```
public class StringBufferandBuilder {
    public StringBuffer contents = new StringBuffer();
    public StringBuilder sbu = new StringBuilder();

    public void log(String message){
        for(int i=0;i<10;i++){
            contents.append(i);
            contents.append("\n");
            sbu.append(i);
            sbu.append("\n");
        }
    }

    public void getcontents(){
        System.out.println("start print StringBuffer");
        System.out.println(contents);
        System.out.println("end print StringBuffer");
    }

    public void getcontents1(){
        System.out.println("start print StringBuilder");
        System.out.println(sbu);
        System.out.println("end print StringBuilder");
    }

    public static void main(String[] args) throws InterruptedException {
        StringBufferandBuilder ss = new StringBufferandBuilder();
        runthread t1 = new runthread(ss,"love");
        runthread t2 = new runthread(ss,"apple");
        runthread t3 = new runthread(ss,"egg");
        t1.start();
        t2.start();
        t3.start();
        t1.join();
        t2.join();
        t3.join();
    }
}

class runthread extends Thread{
    String message;
    StringBufferandBuilder buffer;
```

```

public runthread(StringBufferandBuilder buffer,String message){
    this.buffer = buffer;
    this.message = message;
}
public void run(){
    while(true){
        buffer.log(message);
        //buffer.getcontents();
        buffer.getcontents1();
        try {
            sleep(5000000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
}

```

代码清单 3-91 StringBuffer 和 StringBuilder 对比实验运行输出

```

start print StringBuffer
0123456789
end print StringBuffer
start print StringBuffer
start print StringBuilder
01234567890123456789
end print StringBuffer
start print StringBuilder
01234567890123456789
01234567890123456789
end print StringBuilder
end print StringBuilder
start print StringBuffer
012345678901234567890123456789
end print StringBuffer
start print StringBuilder
012345678901234567890123456789
end print StringBuilder

```

可以看出，StringBuilder 数据并没有按照预想的方式。

StringBuilder 和 StringBuffer 的扩充策略是构造器会创建一个默认大小（通常是 16）的字符数组。在使用中，如果超出这个大小，就会重新分配内存，创建一个更大的数组，并将原先的数组复制过来，再丢弃旧的数组。将原有的容量大小翻倍，以新的容量申请内存空间，建立新的 char 数组，然后将原数组中的内容复制到这个新的数组中。因此，对于较大对象的扩容会涉及大量的内存复制操作。如果能够预先评估大小，会提高性能。

StringBuffer 的构造器会创建一个默认大小（通常是 16）的字符数组。在使用中，如果超出这

个大小，就会重新分配内存，创建一个更大的数组，并将原先的数组复制过来，再丢弃旧的数组。在大多数情况下，你可以在创建 `StringBuffer` 的时候指定大小，这样就避免了在容量不够的时候自动增长，以提高性能。

如代码 3-92 所示，我们为 `StringBuffer` 提供设定的大小。

代码清单 3-92 设定 `StringBuffer` 的大小

```
public class StringBufferDemo{
    void method () {
        stringbuffer buffer = new stringbuffer(max);
        buffer.append ("hello");
    }
    private final int max = 100;
}
```

`StringBuffer` 与 `StringBuilder` 的区别在于，`java.lang.StringBuffer` 是线程安全的可变字符序列，它是一个类似于 `String` 的字符串缓冲区，但是不支持修改。`StringBuilder` 与该类相比，通常应该优先使用 `StringBuilder` 类，因为她支持所有相同的操作，但由于她不执行同步，所以速度更快。为了获得更好的性能，在构造 `StringBuffer` 或 `StringBuilder` 时应尽量指定她的容量。当然如果不超过 16 个字符时就不用了。一般来说，相同场景情况下，我们使用 `StringBuilder` 比使用 `StringBuffer` 仅能获得 10%~15% 的性能提升，但却要冒多线程不安全的风险。综合考虑还是建议使用 `StringBuffer`。

如果你是依靠 Java 编译器来隐式生成实例的话，那么编译的效果几乎和是否使用了 `StringBuilder` 实例毫无关系。每次 CPU 的循环的时间都白白耗费在 GC 或者为 `StringBuilder` 分配默认空间上。一般来说，使用 `StringBuilder` 的效果要优于使用+操作符。如果可能的话请在需要跨多个方法传递引用的情况下选择 `StringBuilder`，因为 `String` 要消耗额外的资源，而 `StringBuilder` 总体来说要优于 `StringBuffer`。

3.3.8 正则表达式不是万能的

正则表达式给我们的印象是非常快，如果万不得已非要在计算密集型代码中使用正则表达式的话，至少要将 `Pattern` 缓存下来，避免反复编译 `Pattern`。

```
static final Pattern HEAVY_REGEX = Pattern.compile("((X)*Y)*Z)*");
```

如果仅使用到了例如 `String[] parts = ipAddress.split("\\.");` 这样简单的正则表达式的话，最好还是用普通的 `char[]` 数组或者是基于索引的操作，比如下面这段可读性比较差的代码其实起到了相同的作用。

代码清单 3-93 替换正则表达式代码

```
int length = ipAddress.length();
int offset = 0;
int part = 0;
for (int i = 0; i < length; i++) {
    if (i == length - 1 ||
        ipAddress.charAt(i + 1) == '.') {
```

```

        parts[part] =
            ipAddress.substring(offset, i + 1);
        part++;
        offset = i + 2;
    }
}

```

虽然与 `split()` 方法相比较, 这段代码的可维护性比较差, 但是它们的执行效率一样, 所以说也并不是非得优化, 当然这最终还是要看程序员的理解了。

3.4 引用类型概念

我在网上看到过某位程序员的留言: “之所以想学习一下 Java 的几种引用类型, 原因有两个:

- (1) 理解 Java Cache 实现、学习 Java 引用与 Java 垃圾回收机制的关系, 内存资源是有限的, 需要合理的利用。Cache 不是仅仅 HashMap 那么简单, Java 引用与 Java 垃圾回收机制也有非常紧密的关系。
- (2) 避免对 Java 引用的错误使用, 某个同事把 5000+ 交易数据放到一个 HashMap 里面, 用一个 Spring Singleton Bean 的全局属性指向该 HashMap。大量运用这种技术, 很快就报 out of memory。再大的内存也架不住对内存的错误使用。理解原理有助于我们尽量少犯或不犯低级错误。”

这一小节就被用来回答网友的这个问题。

当 Java 虚拟机觉得内存不够用的时候, 会触发垃圾回收操作 (GC), 清除无用的对象, 释放内存。可是如何判断一个对象是否是垃圾呢? 其中的一个方法是计算指向该对象的引用数量, 如果引用数量为 0, 那么该对象就为垃圾 (Thread 对象是例外), 否则还有用处, 不能被回收。但是如果把引用数为 0 的对象都回收了, 还是不能满足内存需求怎么办? 我们在这一节简单描述一下, 具体会在第 7 章详细深入讨论。

在很多时候, 一个对象并不是从根部直接引用的, 而是一个对象被其他对象引用, 甚至同时被几个对象所引用, 从而构成一个以根集为顶的树形结构。在 JDK1.2 以前的版本中, 当一个对象不被任何变量引用, 那么程序就无法再使用这个对象。也就是说, 只有对象处于可触及状态, 程序才能使用它。这就像在日常生活中, 从商店购买了某样物品后, 如果有用, 就一直保留它, 否则就把它扔到垃圾箱, 由清洁工人收走。一般说来, 如果物品已经被扔到垃圾箱, 想再把它捡回来使用就不可能了。但有时候情况并不这么简单, 你可能会遇到类似鸡肋一样的物品, 食之无味, 弃之可惜。这种物品现在已经无用了, 保留它会占空间, 但是立刻扔掉它也不划算, 因为也许将来还会派用场。对于这样的可有可无的物品, 一种折中的处理办法是: 如果家里空间足够, 就先把它保留在家里, 如果家里空间不够, 即使把家里所有的垃圾清除, 还是无法容纳那些必不可少的生活用品, 那么再扔掉这些可有可无的物品。

垃圾收集可能是使您感到难于理解的较难的概念之一, 因为它并不能总是毫无遗漏地解决 Java 运行时环境中堆管理的问题。假设我们正面遭遇了内存耗尽的错误, 我们在尝试了一些工具检测没有发现问题时, 我们很容易想到另外一个比较可信的原因, 即这是 Java 虚拟机堆管理的问

题，而不会认为这是自己的程序的缘故。但是事实是，Java 虚拟机并不存在任何被证实的对象泄漏问题。实践证明，垃圾收集器一般能够精确地判断哪些对象可被收集，并且重新收回它们的内存空间给 Java 虚拟机。所以，如果我们遇到了内存耗尽的错误，那么这完全可能是由我们的程序造成的，也就是说程序中存在着“无意识的对象保留 (unintentional object retention)”。

内存泄漏和无意识的对象保留的区别是什么呢？对于用 Java 语言编写的程序来说，确实没有区别。两者都是指在您的程序中存在一些对象引用，但实际上我们并不需要引用这些对象。一个典型的例子是向一个集合中加入一些对象以便以后使用它们，但是却忘了在使用完以后从集合中删除这些对象。因为集合可以无限制地扩大，并且从来不会变小，所以当我们在集合中加入了太多的对象（或者是有很多的对象被集合中的元素所引用）时，就会因为堆的空间被填满而导致内存耗尽的错误。垃圾收集器不能收集这些我们认为已经用完的对象，因为对于垃圾收集器来说，应用程序仍然可以通过这个集合在任何时候访问这些对象，所以这些对象是不可能被当作垃圾的。

对于没有垃圾收集的语言来说，例如 C++，内存泄漏和无意识的对象保留是有区别的。C++ 程序跟 Java 程序一样，可能产生无意识的对象保留。但是 C++ 程序中存在真正的内存泄漏，即应用程序无法访问一些对象以至于被这些对象使用的内存无法释放且返还给系统。令人欣慰的是，在 Java 程序中，这种内存泄漏是不可能出现的。所以，我们更喜欢用“无意识的对象保留”来表示这个令 Java 程序员抓破头皮的内存问题。这样，我们就能区别于其他使用没有垃圾收集语言的程序员。

那么当发现了无意识的对象保留该怎么办呢？首先，需要确定哪些对象是被无意保留的，并且需要找到究竟是哪些对象在引用它们。然后必须安排好应该在哪里释放它们。最容易的方法是使用能够对堆产生快照的检测工具来标识这些对象，比较堆的快照中对象的数目，跟踪这些对象，找到引用这些对象的对象，然后强制进行垃圾收集。有了这样一个检测器，接下来的工作相对而言就比较简单了：

- (1) 等待直到系统达到一个稳定的状态，这个状态下大多数新产生的对象都是暂时的，符合被收集的条件；这种状态一般在程序所有的初始化工作都完成了之后。
- (2) 强制进行一次垃圾收集，并且对此时的堆做一份对象快照。
- (3) 进行任何可以产生无意地保留的对象的操作。
- (4) 再强制进行一次垃圾收集，然后对系统堆中的对象做第二次对象快照。

比较两次快照，看看哪些对象的被引用数量比第一次快照时增加了。因为在快照之前强制进行了垃圾收集，那么剩下的对象都应该被应用程序所引用的对象，并且通过比较两次快照我们可以准确地找出那些被程序保留的、新产生的对象。根据我们对应用程序本身的理解，并且根据对两次快照的比较，判断出哪些对象是被无意保留的。跟踪这些对象的引用链，找出究竟是哪些对象在引用这些无意地保留的对象，直到您找到了那个根对象，它就是产生问题的根源。

Java 中提供了 4 个级别的引用，即强引用 (FinalReference)、软引用 (SoftReference)、弱引用 (WeakReference)、虚引用 (PhantomReference) 这四个级别。在这 4 个级别中只有强引用类是包内可见的，其他 3 种引用类型均为 public，可以在应用程序中直接使用，垃圾回收器会尝试回收只有弱引用的对象。

- 强引用 (Strong Reference)：在一个线程内，无须引用直接可以使用的对象，强引用不会

垃圾回收器可能采取不同的算法来判断对象的引用是否存在。一个常见的做法是使用引用计数器。当有新的引用指向某个对象时，把该计数器的值加 1；当一个引用失效时，就把该计数器的值减 1。例如，显式地把一个引用某个对象的值变为 0 的时候，说明不存在任何指向该对象的引用，该对象可以被垃圾回收器回收。引用计数器的原理比较简单，但是实现起来需要编译器的支持，另外使用引用计数器不能解决循环引用孤岛的回收问题。比如，三个对象之间互相存在引用关系，但是并不存在指向这三个对象的其他引用，这三个对象实际上就成为了内存区域中的一个孤岛。这三个对象的引用计数器的值都不为 0，因此无法通过引用计数器的方式来回收。

由于引用计数器存在无法处理“孤岛”的问题，Java 虚拟机的垃圾回收器没有采用这种做法，而是采取跟踪对象引用的做法。这种做法会从虚拟机内存中的某些存活对象开始，递归检查这些对象所引用的其他对象，知道找到不引用其他对象的对象为止。在这个过程中所发现的所有对象都会被标记为存活的，而其他对象则是可以被回收的。这个遍历过程的起始对象是一个集合，称之为根集合。根集合中一般包括系统类、程序寄存器、JNI 全局引用、静态变量和线程的当前活动栈中的变量所指向的对象等。可以将这个跟踪过程看成是基于引用关系的树的遍历。在跟踪过程中发现的存活对象被称为可达的。将从遍历的根节点到当前存活对象的路径称为可达路径。这条路径的边对应的是对象之间的引用。如果一个对象的可达路径中只包含强引用，则把这个对象成为强引用可达的。程序中的大多数存活对象都是强引用可达的。

对于垃圾回收器来说，强引用的存在会阻止一个对象被回收。在垃圾回收器遍历对象引用并进行标记之后，如果一个对象是强引用可达的，那么这个对象不会作为垃圾回收的候选。因为该对象仍然被程序所使用，回收其内存显然是一个错误的做法。虽然由于垃圾回收器的存在，Java 虚拟机中并不存在真正意义上的内存泄漏，但是某些错误的用法会对程序中所能使用的内存空间造成影响。这些情况可以看成是另外一种意义上的内存泄露，这些内存泄露的发生也都和强引用的使用有关。

总的来说，在我们的应用程序代码中使用的大部分引用实际上都是强引用，强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。当内存空间不足，Java 虚拟机宁愿抛出 `OutOfMemoryError` 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。

比如下面这段代码中的 `object` 和 `str` 都是强引用：

```
2Object object = new Object();
String str = "hello";
```

前面说过，只要某个对象有强引用与之关联，JVM 必定不会回收这个对象，即使在内存不足的情况下，JVM 宁愿抛出 `OutOfMemory` 错误也不会回收这种对象，比如清单 3-97 所示代码。

代码清单 3-97 强引用示例

```
public class Main {
    public static void main(String[] args) {
        new Main().fun1();
    }

    public void fun1() {
```



```

        Object object = new Object();
        Object[] objArr = new Object[1000]; //当运行至 Object[] objArr = new
Object[1000];这句时,如果内存不足,JVM会抛出 OOM 错误也不会回收 object 指向的对象。
    }
}

```

当 fun1 运行完之后,object 和 objArr 都已经不存在了,所以它们指向的对象都会被 JVM 回收。如果想中断强引用和某个对象之间的关联,可以显示地将引用赋值为 null,这样一来 JVM 在合适的时间就会回收该对象。比如 Vector 类的 remove 方法中就是通过将引用赋值为 null 来实现清理工作的。

代码清单 3-98 Vector 源码

```

/**
 * Removes the element at the specified position in this Vector.
 * Shifts any subsequent elements to the left (subtracts one from their
 * indices). Returns the element that was removed from the Vector.
 *
 * @throws ArrayIndexOutOfBoundsException if the index is out of range
 *         (({@code index < 0 || index >= size()}))
 * @param index the index of the element to be removed
 * @return element that was removed
 * @since 1.2
 */
public synchronized E remove(int index) {
    modCount++;
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);
    Object oldValue = elementData[index];

    int numMoved = elementCount - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
            numMoved);
    elementData[--elementCount] = null; //GC 回收对象

    return (E)oldValue;
}

```

在实现一个缓存系统的时候,如果全部使用强引用,那么你需要自己去手动的把某些引用 Clear 掉(引用置为 Null),否则迟早会抛出 Out Of Memory 错误。缓存系统引入弱引用或者软引用的唯一原因是,把引用 Clear 的事情交由 Java 垃圾回收器来处理,Cache 程序自己置身事外。

通常来说,应用程序内部的内存泄露有两种情况。一种是虚拟机中存在程序无法使用的内存区域。这些内存区域被程序中一些无法使用的存活对象占用。这些对象由于存在隐式的强引用,无法对其进行垃圾回收。但是在程序的正常运行过程中,这些对象也无法被使用。造成这种问题的原因通常是程序编写时的逻辑错误。另一种情况是程序中存在大量存活时间过长的对象。这些对象的存活时间长于使用它们的对象。在正常情况下,这些对象在引用它们的对象被回收之后,

也应该被回收，但是由于某些程序中的错误而没有被回收。这些对象无法被回收，仍然占据着虚拟机中的内存资源。时间长了，虚拟机会因为没有足够的内存分配给新的对象而抛出 OOM 错误。

第一种情况出现的实例代码如下所示。清单中给出了一个简单的先进先出队列的实现。它在内部使用了一个 `java.util.List` 接口的实现对象来保存队列中的对象。向队列中添加新的对象会被直接放在 `List` 接口实现对象的末尾。队列的队首位置则由内部变量 `topIndex` 来维护。每次有对象被移出队列时，`topIndex` 的值会增加 1。这个队列实现的问题在于出队列的方法只简单地改变了 `topIndex` 的值，并没有把对象从队列中剔除。在经过若干次队列操作之后，`topIndex` 的值会逐渐变大。变量 `backendList` 所指向的对象中包含的序号小于 `topIndex` 的对象无法被队列的使用者通过正常的方式来访问。由于 `backendList` 所指向的对象仍然包含指向这些对象的强引用，因此这些对象也无法被垃圾回收，这些对象占用的内存就成为虚拟机中无法使用的区域。

代码清单 3-99 先进先出队列示例

```
import java.util.ArrayList;
import java.util.List;

public class LeakingQueue<T> {
    private List<T> backendList = new ArrayList<T>();
    private int topIndex = 0;
    public void enqueue(T value){
        backendList.add(value);
    }

    public T dequeue(){
        T result = backendList.get(topIndex);
        topIndex++;
        return result;
    }

    public static void main(String[] args){
        LeakingQueue lq = new LeakingQueue();
        for(int i=0;i<10000000;i++){
            for(int j=0;j<10000000;j++){
                lq.enqueue(i);
                lq.dequeue();
            }
        }
    }
}
```

输出如清单 3-100 所示。

代码清单 3-100 先进先出队列示例运行输出

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.Arrays.copyOf(Unknown Source)
    at java.util.Arrays.copyOf(Unknown Source)
```



```

at java.util.ArrayList.grow(Unknown Source)
at java.util.ArrayList.ensureExplicitCapacity(Unknown Source)
at java.util.ArrayList.ensureCapacityInternal(Unknown Source)
at java.util.ArrayList.add(Unknown Source)
at LeakingQueue.enqueue(LeakingQueue.java:8)
at LeakingQueue.main(LeakingQueue.java:21)

```

第二种情况的典型情景发生在使用基于内存实现的缓存的时候。如下代码利用 `calculate` 方法进行实际运算所需的时间可能比较长，因此使用了一个 `java.util.HashMap` 类的对象来保存之前计算的结果。在 `calculate` 方法被调用时，会先检查缓存中是否已经存在之前计算出来的结果，这样可以避免重复的计算，进而提高性能。不过这种做法延长了计算结果对象的存活时间。在不使用缓存的情况下，在 `calculate` 方法的调用者获得计算结果对象，并完成对该对象的使用之后，就可以对该对象进行垃圾回收。当使用了缓存之后，计算结果对象的存活时间就变得至少和用来进行缓存的 `HashMap` 类的对象一样长。因为 `HashMap` 类的对象有所包含的所有计算结果对象的引用，所以，只要 `HashMap` 类的对象无法被回收，其中所包含的计算结果对象也无法被回收。在 `calculate` 方法被多次调用之后，缓存中包含的对象会越来越多，导致占用的内存越来越大，而程序中其他部分可用的内存则越来越少。

代码清单 3-101 使用 HashMap 保存计算结果方法

```

import java.util.HashMap;
import java.util.Map;

public class Calculator {
    private Map<String, Object> cache = new HashMap<String, Object>();
    public Object calculate(String expr) {
        if (cache.containsKey(expr)) {
            return cache.get(expr);
        }
        Object result = doCalculate(expr);
        cache.put(expr, result);
        return result;
    }

    private Object doCalculate(String expr) {
        return new Object();
    }

    public static void main(String[] args) {
        Calculator cal = new Calculator();
        for (int i = 0; i < 100000000; i++) {
            cal.calculate(String.valueOf(i));
        }
    }
}

```

代码 3-101 的运行输出如清单 3-102 所示。

代码清单 3-102 代码 3-95 运行输出

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
  at java.lang.Integer.toString(Unknown Source)
  at java.lang.String.valueOf(Unknown Source)
  at Calculator.main(Calculator.java:23)
```

从前面两个示例可以看出，强引用所提供的与垃圾回收器的交互功能非常有限。当强引用存在的时候，所指向的对象无法被垃圾回收。为了增强程序与垃圾回收器的交互能力，JDK1.2 引入了 `java.lang.ref` 包，提供了 3 种新的引用类型，分别是软引用、弱引用和虚引用。这些引用类型除了可以引用对象之外，还可以在不同程度上影响垃圾回收器对被引用对象的处理行为。

3.4.2 软引用 (Soft Reference)

首先，我们假设存在一个用户信息查询系统的应用场景，用户信息查询系统负责查询存储在磁盘文件或者数据库中的用户档案信息。作为一个该应用程序的使用者，我们完全有可能需要回头去查看几分钟甚至几秒钟前查看过的用户信息（同样，我们在浏览 Web 页面的时候也经常会使用“后退”按钮）。这时我们通常会有两种程序实现方式，一种是把过去查看过的用户信息保存在内存中，每一个存储了用户信息的 Java 对象的生命周期贯穿整个应用程序始终，另一种是当使用着开始查看其他用户信息的时候，把存储了当前所查看的用户信息的 Java 对象结束引用，使得垃圾收集线程可以回收其所占用的内存空间，当用户再次需要浏览该用户信息的时候，重新构建该用户的信息。很显然，第一种实现方法将造成大量的内存浪费，而第二种实现的缺陷在于即使垃圾收集线程还没有进行垃圾收集，包含用户信息的对象仍然完好地保存在内存中，应用程序也要重新构建一个对象。从第 2 章我们知道，访问磁盘文件、访问网络资源、查询数据库等操作都是影响应用程序执行性能的重要因素，如果能重新获取那些尚未被回收的 Java 对象的引用，必将减少不必要的访问，大大提高程序的运行速度。

软引用是除了强引用外最强的引用类型，我们可以通过 `java.lang.ref.SoftReference` 使用软引用。一个持有软引用的对象，它不会被 JVM 很快回收，JVM 会根据当前堆的使用情况来判断何时回收。当堆使用率临近阈值时，才会去回收软引用的对象。只要有足够的内存，软引用便可能在内存中存活相当长一段时间。因此，就如上一节所说的，软引用可以用于实现对内存敏感的 Cache。垃圾回收器会保证在抛出 OOM 错误之前，回收掉所有软引用可达的对象。通过软引用，垃圾回收器就可以在内存不足时释放软引用可达的对象所占的内存空间。程序所要做的是保证软引用可达的对象被垃圾回收器回收之后，程序也能正常工作。在之前介绍的图像编辑器打开文件的示例中，可以用强引用指向当前正在编辑的图片，而用软引用指向不处于编辑状态的其他已经打开的图片。这样当图像编辑器程序的内存不足时，垃圾回收器可以释放不处于编辑状态的图片所占的内存空间。当程序中需要引用所占内存比较大的对象时，可以考虑使用软引用来指向该对象。

软引用的特点是它的一个实例保存对一个 Java 对象的软引用，该软引用的存在不妨碍垃圾收集线程对该 Java 对象的回收。也就是说，一旦软引用保存了对一个 Java 对象的软引用后，在垃圾线程对这个 Java 对象回收前，软引用类所提供的 `get()` 方法返回 Java 对象的强引用。另外，一旦垃圾线程回收该 Java 对象之后，`get()` 方法将返回 `null`。

代码清单 3-103 软引用实例

```

import java.lang.ref.Reference;
import java.lang.ref.ReferenceQueue;
import java.lang.ref.SoftReference;

public class MyObject {
    public static ReferenceQueue softQueue;

    @Override
    protected void finalize() throws Throwable{
        super.finalize();
        Reference<MyObject> obj = null;
        try{
            obj = (Reference<MyObject>)softQueue.remove();
        }catch(InterruptedException e){
            e.printStackTrace();
        }
        System.out.println(obj);
        if(obj!=null){
            System.out.println("Object for SoftReference is"+obj.get());
        }
        System.out.println("MyObject's finalize called");//只有当 GC 回收时才会输出
    }

    @Override
    public String toString(){
        return "I am MyObject";
    }

    public static void main(String[] args){
        MyObject obj = new MyObject();//强引用
        softQueue = new ReferenceQueue<MyObject>();//创建引用那个队列
        SoftReference<MyObject> softRef = new SoftReference<MyObject>(obj,softQueue);
        //创建软引用
        System.out.println(obj);
        obj=null;//删除强引用
        System.gc();
        System.out.println(obj);
        System.out.println("After GC:Soft Get= "+softRef.get());
        System.out.println("分配大块内存");
        byte[] b = new byte[1000*1024*925];//分配一块较大内存区,强迫 GC
        System.out.println("After new byte[]:Soft Get= "+softRef.get());
        System.out.println(obj);
    }
}

```

上面的例子中,首先构造 `MyObject` 对象,并将其赋值给 `obj` 变量,构成强引用。然后使用 `SoftReference` 构造这个 `MyObject` 对象的软引用 `softRef`,并注册到 `softQueue` 引用队列。当 `softRef`

被回收时，会被加入 `softQueue` 队列。设置 `obj=null`，删除这个强引用，因此系统内对 `MyObject` 对象的引用只剩下软引用。此时，显示调用 GC，通过软引用的 `get()` 方法，取得 `MyObject` 对象实例的强引用，发现对象并未被回收。这说明 GC 在内存充足的情况下不会回收入软引用对象。接着分配一块大的堆空间，把应用最大内存设置为 `1MB(Xmx1M)`，这样会使系统堆空存使用紧张，从而产生新一轮 GC。在这次 GC 后，`softRef.get()` 不再返回 `MyObject` 对象，而是返回 `null`，这说明系统内存紧张下 JVM 会回收软引用。软引用被回收时，会被加入注册的引用队列。

作为一个 Java 对象，软引用对象除了具有保存软引用的特殊性之外，也具有 Java 对象的一般性。所以，当软引用对象被回收之后，虽然这个软引用对象的 `get()` 方法返回 `null`，但这个对象已经不再具有存在的价值，需要一个适当的清除机制，避免大量软引用对象带来的内存泄漏。在 `java.lang.ref` 包里还提供了 `ReferenceQueue`。如果在创建软引用对象的时候，使用了一个 `ReferenceQueue` 对象作为参数提供给软引用的构造方法，如清单 3-104 所示。

代码清单 3-104 ReferenceQueue 对象

```
ReferenceQueue queue = new ReferenceQueue();
SoftReference ref=new SoftReference(myObject, queue);
```

那么当这个 `SoftReference` 所持有的软引用的 `myObject` 被垃圾收集器回收的同时，`ref` 所强引用的软引用对象被列入 `ReferenceQueue`。也就是说，`ReferenceQueue` 中保存的对象是 `Reference` 对象，而且是已经失去了它所软引用的对象的 `Reference` 对象。另外从 `ReferenceQueue` 这个名字也可以看出，它是一个队列，当我们调用它的 `poll()` 方法的时候，如果这个队列中不是空队列，那么将返回队列前面的那个 `Reference` 对象。

在任何时候，我们都可以调用 `ReferenceQueue` 的 `poll()` 方法来检查是否有它所关心的非强可及对象被回收。如果队列为空，将返回一个 `null`，否则该方法返回队列中前面的一个 `Reference` 对象。利用这个方法，我们可以检查哪个 `SoftReference` 所软引用的对象已经被回收。于是我们可以把这些失去所软引用的对象的 `SoftReference` 对象清除掉。常用的方式如 `SoftReference ref = null;while ((ref = (EmployeeRef) q.poll()) != null) { // 清除 ref}`。

下面代码中的 `FileEditor` 类用来对多个文件进行编辑。出于性能方面的考虑，`FileEditor` 类的对象会在内部缓存之前已经打开过的文件的数据内容，以方便用户在同时打开的多个文件之间进行快速切换。同时打开的文件过多会占用比较多的内存资源。因此，表示文件数据的 `FileData` 类使用软引用用来指向包含文件数据的 `byte[]` 对象。当虚拟机的内存不足时，这些 `byte[]` 对象可以被垃圾回收器释放。这里需要注意 `FileData` 类的 `getData` 方法的实现中对软引用的使用方式。通过 `get` 方法获取软引用所指向的对象之后，需要判断这个对象是否还存活。如果 `get` 方法的返回值为 `null`，那么说明对该对象的引用已经被清空，应该重新创建出相关的对象。

代码清单 3-105 文件编辑器示例

```
import java.io.IOException;
import java.lang.ref.SoftReference;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.HashMap;
```



```
import java.util.Map;

public class FileEditor {
    private static class FileData{
        private Path filePath;
        private SoftReference<byte[]> dataRef;

        public FileData(Path filePath){
            this.filePath = filePath;
            this.dataRef = new SoftReference<byte[]>(new byte[0]);
        }

        public Path getPath(){
            return filePath;
        }

        public byte[] getData() throws IOException{
            byte[] dataArray = dataRef.get();
            if(dataArray == null || dataArray.length == 0){
                dataArray = readFile();
                dataRef = new SoftReference<byte[]>(dataArray);
                dataArray = null;
            }
            return dataRef.get();
        }

        private byte[] readFile() throws IOException{
            return Files.readAllBytes(filePath);
        }
    }

    private FileData currentFileData;
    private Map<Path,FileData> openedFiles = new HashMap<>();

    public void switchTo(String filePath){
        Path path = Paths.get(filePath).toAbsolutePath();
        if(openedFiles.containsKey(path)){
            currentFileData = openedFiles.get(path);
        }else{
            currentFileData = new FileData(path);
            openedFiles.put(path,currentFileData);
        }
    }

    public void useFile() throws IOException{
        if(currentFileData != null){
            System.out.println(String.format("",currentFileData.getPath(),
```

```

currentFileData.getData().length));
    }
}

public static void main(String[] args){
    FileEditor fe = new FileEditor();
    try {
        for(int i=0;i<100;i++){
            fe.switchTo("你的文件夹");
            fe.useFile();
        }
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
}

```

使用 `FileEditor` 类的对象依次打开某个目录下包含的大小各异多个文件，同时通过虚拟机的启动参数 “-Xmx” 把虚拟机所用的堆内存的最大值设置为一个相对较小的值。在运行时会发现，虽然虚拟机可用堆内存的最大值远小于所处理的所有文件的大小的总和，但是程序在运行中也不会抛出 OOM 错误。这是因为当虚拟机中内存不足时，软引用指向的 `byte[]` 对象会被释放，从而可以腾出内存空间供之后的文件操作使用。如果使用强引用指向 `byte[]` 对象，在打开目录下的部分文件之后，就会出现 OOM 错误，这是因为虚拟机的堆内存不足以容纳全部文件的内容，而垃圾回收器又无法释放强引用指向的 `byte[]` 对象。

3.4.3 弱引用 (Weak Reference)

弱引用表示对一个对象的引用，使用弱引用后，可以维持对 `referent` 的引用，而不会阻止它被垃圾收集。当垃圾收集器跟踪堆的时候，如果对一个对象的引用只有弱引用，那么这个 `referent` 就会成为垃圾收集的候选对象，就像没有任何剩余的引用一样，而且所有剩余的弱引用都被清除。弱引用是在构造时设置的，在没有被清除之前，可以获取它的值，如果弱引用被清除了，无论是被垃圾收集了，还是有人调用了 `clear()` 方法，它都会返回 `null`。所以，我们在使用弱引用对象之前，都需要检查是否返回一个非 `null` 值。

弱引用是一种比软引用较弱的引用类型。在系统 GC 时，只要发现弱引用，不管系统堆空间是否足够，都会将对象进行回收。但是，由于垃圾回收器的线程通常优先级很低，因此并不一定能很快地发现持有弱引用的对象。在这种情况下，弱引用对象可以存在较长的时间。一旦一个弱引用对象被垃圾回收器回收，便会加入到一个注册引用队列中。上述例子只需要更改对象类型为 `weakReference` 即可，会看到一旦 `obj` 被设置为 `null`，GC 立即回收若类型实例。

```
WeakReference<MyObject> Ref = new WeakReference<MyObject>(obj,softQueue); //创建弱引用
```

弱引用、软引用都非常适合来保存那些可有可无的缓存数据。如果这么做，当系统内存不足时，这些缓存数据会被回收，不会导致内存溢出。当内存资源充足时，这些缓存数据又可以存在相当长的时间，从而起到加速系统的作用。

用一个普通的强引用拷贝一个对象引用时，限制对象的生命周期至少与被拷贝的引用的生命周期一样长。如果我们将一个对象放入一个全局集合中的话，那么它可能就和程序的生命周期一样。另一方面，在创建对一个对象的弱引用时，完全没有扩展对象的生命周期，只是在对象仍然存活的时候，保持另一种到达它的方法。

弱引用的重要作用是解决对象的存活时间过长的问题。在程序中，一个对象的实际存活时间应该与它的逻辑存活时间一样。从逻辑上来说，一个对象应该在某个方法调用完成之后就不再被需要了，可以对其进行垃圾回收。但是，如果仍然有其他的强引用存在，该对象的实际存活时间会长于逻辑存活时间，直到其他的强引用不再存在。这样的对象在程序中过多出现会导致虚拟机的内存占用上升，最后产生 OOM 错误。要解决这样的问题，需要小心注意管理对象上的强引用。当不再需要引用一个对象时，显式地清除这些强引用。不过这会对开发人员提出更高的要求。更好的方法是使用弱引用替换强引用用来引用这些对象。这样既可以引用对象，又可以避免强引用带来的问题。

比较典型的例子是在哈希表中使用弱引用。如下面代码所示，通过 `BookKeeper` 类来对图书及其借阅者进行管理。在内部实现中使用了一个 `HashMap` 类的对象来保存图书及其借阅者之间的对应关系。由于 `HashMap` 类的对象具有对所包含的键和值得对象的强引用，这会使 `Book` 类和 `User` 类对象的存活时间变得至少和 `HashMap` 类的对象本身一样长。当 `HashMap` 类的对象本身还存活时，其中所包含的 `Book` 类和 `User` 类的对象都无法被垃圾回收器回收。

代码清单 3-106 哈希表中使用弱引用示例

```
import java.util.Set;
import org.apache.hadoop.hbase.security.User;

public class BookKeeper {
    private Map<Book, Set<User>> books = new HashMap<>();

    public void borrowBook(Book book, User user) {
        Set<User> users = null;
        if (books.containsKey(book)) {
            users = books.get(book);
        } else {
            users = new HashSet<User>();
            books.put(book, users);
        }
        users.add(user);
    }

    public void returnBook(Book book, User user) {
        if (books.containsKey(book)) {
            Set<User> users = books.get(book);
            users.remove(user);
        }
    }
}
```

解决这个问题的做法是使用弱引用来指向这些对象，而不是使用默认的强引用。因为这样使用 `Map` 接口的情况很多，Java 标准库提供了 `java.util.WeakHashMap` 类来满足这种常见的需求。

`WeakHashMap` 类使用弱引用来指向其中所包含的键，而键对应的值对象仍然由强引用来指向。使用弱引用的好处是 `WeakHashMap` 类的对象本身对其中所包含的键的弱引用不会影响键对象的垃圾回收。当键对象不存在其他类型更强的引用时，键对象会被从 `WeakHashMap` 类的对象中删除。上面的代码需要把 `books` 对应的 `Map` 接口的实现类换成 `WeakHashMap` 类既可。不过 `WeakHashMap` 类的对象中包含的值对象仍然由强引用来指向，因此不能在值对象中包含键对象的引用。这种循环引用会导致对象无法被垃圾回收器回收。如果觉得对于值对象使用强引用不合适，可以在添加到 `WeakHashMap` 类的对象之前用一个 `WeakReference` 类的对象来包装它。

前面提到过无意识对象保留概念，无意识对象保留最常见的原因是使用 `Map` 将元数据与临时对象（transient object）相关联。假定一个对象具有中等生命周期，比分配它的那个方法调用的生命周期长，但是比应用程序的生命周期短，如客户机的 `Socket` 连接。需要将一些元数据与这个 `Socket` 关联，如生成连接的用户的标识。在创建 `Socket` 时是不知道这些信息的，并且不能将数据添加到 `Socket` 对象上，因为不能控制 `Socket` 类或者它的子类。这时，典型的方法就是在一个全局 `Map` 中存储这些信息，如代码清单 3-107 中的 `SocketManager` 类所示。

代码清单 3-107 使用一个全局 `Map` 将元数据关联到一个对象

```
public class SocketManager {
    private Map<Socket,User> m = new HashMap<Socket,User>();
    public void setUser(Socket s, User u) {
        m.put(s, u);
    }
    public User getUser(Socket s) {
        return m.get(s);
    }
    public void removeUser(Socket s) {
        m.remove(s);
    }
}

SocketManager socketManager;
socketManager.setUser(socket, user);
```

这种方法的问题是元数据的生命周期需要与套接字的生命周期挂钩，但是除非准确地知道什么时候程序不再需要这个套接字，并记住从 `Map` 中删除相应的映射，否则，`Socket` 和 `User` 对象将会永远留在 `Map` 中，远远超过响应了请求和关闭 `Socket` 的时间。这会阻止 `Socket` 和 `User` 对象被垃圾收集，即使应用程序不会再使用它们。这些对象留下来不受控制，很容易造成程序在长时间运行后内存爆满。除了最简单的情况，在几乎所有情况下找出什么时候 `Socket` 不再被程序使用是一件很烦人和容易出错的任务，需要人工对内存进行管理。

弱引用对于构造弱集合最有用，如那些在应用程序的其余部分使用对象期间存储关于这些对象的元数据的集合，这就是 `SocketManager` 类所要做的工作。因为这是弱引用最常见的用法，`WeakHashMap` 对键（而不是对值）使用弱引用。如果在一个普通 `HashMap` 中用一个对象作为键，那么这个对象在映射从 `Map` 中删除之前不能被回收，`WeakHashMap` 使您可以用一个对象作为 `Map` 键，同时不会阻止这个对象被垃圾收集。代码清单 3-108 给出了 `WeakHashMap` 的 `get()` 方法的一种可能实现，它展示了弱引用的使用方式。

代码清单 3-108 WeakReference.get()的一种可能实现

```

public class WeakHashMap<K,V> implements Map<K,V> {
    private static class Entry<K,V> extends WeakReference<K>
        implements Map.Entry<K,V> {
        private V value;
        private final int hash;
        private Entry<K,V> next;
        ...
    }
    public V get(Object key) {
        int hash = getHash(key);
        Entry<K,V> e = getChain(hash);
        while (e != null) {
            K eKey= e.get();
            if (e.hash == hash && (key == eKey || key.equals(eKey)))
                return e.value;
            e = e.next;
        }
        return null;
    }
}

```

调用 `WeakReference.get()` 方法时，它返回一个对对象的强引用（如果它仍然存活的话），因此不需要担心映射在 `while` 循环体中消失，因为强引用会防止它被垃圾收集。`WeakHashMap` 的实现展示了弱引用的一种常见用法，一些内部对象扩展 `WeakReference`。在向 `WeakHashMap` 中添加映射时，请记住映射可能会在以后“脱离”，因为键被垃圾收集了。在这种情况下，`get()` 返回 `null`，这使得测试 `get()` 的返回值是否为 `null` 变得比平时更重要了。

代码清单 3-108 中所述的 `SocketManager` 防止泄漏很容易，只要用 `WeakHashMap` 代替 `HashMap` 就行了，如代码清单 3-109 所示。

如果 `SocketManager` 需要线程安全，那么可以用 `Collections.synchronizedMap()` 包装 `WeakHashMap`。当映射的生命周期必须与键的生命周期联系在一起时，可以使用这种方法。不过，应当小心不滥用这种技术，大多数时候还是应当使用普通的 `HashMap` 作为 `Map` 的实现。

代码清单 3-109 用 WeakHashMap 修复 SocketManager

```

public class SocketManager {
    private Map<Socket,User> m = new WeakHashMap<Socket,User>();
    public void setUser(Socket s, User u) {
        m.put(s, u);
    }
    public User getUser(Socket s) {
        return m.get(s);
    }
}

```

`WeakHashMap` 用弱引用承载映射键，这使得应用程序不再使用键对象时它们可以被垃圾收集，`get()` 实现可以根据 `WeakReference.get()` 是否返回 `null` 来区分死的映射和活的映射。但是这只是防止

Map 的内存消耗在应用程序的生命周期中不断增加所需要做的工作的一半，还需要做一些工作以便在键对象被收集后从 Map 中删除死项。否则，Map 会充满对应于被回收键的项。虽然这对于应用程序是不可见的，但是它仍然会造成应用程序耗尽内存，因为即便键被收集了，Map.Entry 和值对象也不会被收集。我们可以通过周期性地扫描 Map，对每一个弱引用调用 get()，并在 get() 返回 null 时删除那个映射而消除死映射。但是如果 Map 有许多活的项，那么这种方法的效率很低。如果有一种方法可以在弱引用的对象被垃圾收集时发出通知就好了，这就是引用队列的作用。

引用队列是垃圾收集器向应用程序返回关于对象生命周期的信息的主要方法。弱引用有两个构造函数：一个只取引用作为参数，另一个还取引用队列作为参数。如果用关联的引用队列创建弱引用，在对象引用成为 GC 候选对象时，这个引用对象就在引用清除后加入到引用队列中。之后，应用程序从引用队列提取引用并了解到它的引用对象已经被收集，因此可以进行相应的清理活动，如去掉已不在弱集合中的对象的项。

WeakHashMap 有一个名为 expungeStaleEntries() 的私有方法，大多数 Map 操作中会调用它，它去掉引用队列中所有失效的引用，并删除关联的映射。代码清单 3-110 展示了 expungeStaleEntries() 的一种可能实现。用于存储键-值映射的 Entry 类型扩展了 WeakReference，因此当 expungeStaleEntries() 要求下一个失效的弱引用时，它得到一个 Entry。用引用队列代替定期扫描内容的方法来清理 Map 更有效，因为清理过程不会触及活的项，只有在有实际加入队列的引用时它才工作。

代码清单 3-110 WeakHashMap.expungeStaleEntries() 的可能实现

```
private void expungeStaleEntries() {
    Entry<K,V> e;
    while ( (e = (Entry<K,V>) queue.poll()) != null) {
        int hash = e.hash;
        Entry<K,V> prev = getChain(hash);
        Entry<K,V> cur = prev;
        while (cur != null) {
            Entry<K,V> next = cur.next;
            if (cur == e) {
                if (prev == e)
                    setChain(hash, next);
                else
                    prev.next = next;
                break;
            }
            prev = cur;
            cur = next;
        }
    }
}
```

我们再通过一些例子来看看 WeakHashMap 类的使用。WeakHashMap 在 java.util 包内，它实现了 Map 接口，是 HashMap 的一种实现，它使用弱引用作为内部数据的存储方案。WeakHashMap 是弱引用的一种典型应用，它可以作为简单的缓存表解决方案。

如果在系统中需要一张很大的 Map 表，Map 中的表项作为缓存之用，这也意味着即使没有能从该 Map 中取得相应的数据，系统也可以通过候选方案获取这些数据。虽然这样会消耗更多的时间，但是不影响系统的正常运行。在这种场景下，使用 WeakHashMap 是最为适合的。因为 WeakHashMap 会在系统内存范围内，保存所有表项，而一旦内存不够，在 GC 时没有被引用的表项也会很快被清除掉，从而避免系统内存溢出。

代码清单 3-111 所示的两段代码分别用 WeakHashMap 和 HashMap 保存大量的数据，JVM 设置 -Xmx1M 的最大可用堆。

代码清单 3-111 WeakHashMap 和 HashMap 对比实验

```
public static void main(String[] args){
    long start = System.currentTimeMillis();
    Map map = new WeakHashMap();
    List list = new ArrayList();
    for(int i=0;i<100000;i++){
        Integer ii = new Integer(i);
        map.put(ii, new byte[i]);
    }
    System.out.println(System.currentTimeMillis()-start);

    start = System.currentTimeMillis();
    map = new HashMap();
    list = new ArrayList();
    for(int i=0;i<100000;i++){
        Integer ii = new Integer(i);
        map.put(ii, new byte[i]);
    }
    System.out.println(System.currentTimeMillis()-start);
}
```

输出结果如代码清单 3-112 所示。

代码清单 3-112 输出结果

```
6880
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at WeakHashMapV$HashMap.main(WeakHashMapV$HashMap.java:24)
```

由此可见，WeakHashMap 会在系统内存紧张时使用弱引用自动释放持有弱引用的内存数据，而 HashMap 是强引用类型，会抛出 OOM。但是在内存充足时 HashMap 的速度更快。

上述代码中只要加一句代码就会让 WeakHashMap 抛出 OOM 错误，即 list.add(ii);。

这行代码对 key 进行了强引用，导致 WeakHashMap 不会自动启动实例回收机制。因此，如果系统在系统中通过 WeakHashMap 自动清理数据，就尽量不要在系统的其他地方强引用 WeakHashMap 的 key，否则这些 key 就不会被回收，WeakHashMap 也就无法正常释放它们所占用的表项。

总的来说，弱引用对象与软引用对象的最大不同就在于，当 GC 在进行回收时，需要通过算

法检查是否回收软引用对象，而对于弱引用对象，GC 总是进行回收。弱引用对象更容易、更快被 GC 回收。虽然，GC 在运行时一定回收 Weak 对象，但是复杂关系的弱对象群常常需要好几次 GC 的运行才能完成。就像上面描述的场景，弱引用对象常常用于 Map 结构中，引用数据量较大的对象，一旦该对象的强引用为 null 时，GC 能够快速地回收该对象空间。

幽灵引用 (Phantom Reference) 是强度最弱的一种引用类型，用 `java.lang.ref.PhantomReference` 类来表示。幽灵引用的主要目的是在一个对象所占的内存被实际回收之前得到通知，从而可以进行一些相关的清理工作。幽灵引用在使用方式上与之前介绍的两种引用类型有很大的不同：首先幽灵引用在创建时必须提供一个引用队列作为参数；其次幽灵引用对象的 `get` 方法总是返回 null，因此无法通过幽灵引用来获取被引用的对象。

幽灵引用在使用的时候只能通过引用队列来操作。幽灵引用的最大优势在于引用对象被添加到队列中的时机。Java 语言提供了对象终止 (finalization) 机制来允许开发人员提供对象被销毁之前的自定义处理逻辑。Object 类提供了 `finalize` 方法来添加自定义的销毁逻辑。如果一个类有特殊的销毁逻辑，可以覆写 `finalize` 方法。从功能上来说，`finalize` 方法与 C++ 中的析构函数比较相似，但是 Java 采用的是基于垃圾回收器的自动内存管理机制，所以 `finalize` 方法在本质上不同于 C++ 中的析构函数。当垃圾回收器发现没有引用指向一个对象时，会调用这个对象的 `finalize` 方法。通常在这个方法中进行一些资源释放和清理的工作，比如关闭文件、套接字和数据库连接等。由于 `finalize` 方法的存在，虚拟机中的对象一般处于三种可能的状态。第一种是可达状态，当有引用指向该对象时，该对象处于可达状态。根据引用类型的不同，有可能处于强引用可达、软引用可达或弱引用可达状态。第二种是可复活状态，如果对象的类覆写了 `finalize` 方法，则对象有可能处于该状态。虽然垃圾回收器是在对象没有引用的情况下才调用其 `finalize` 方法，但是在 `finalize` 方法的实现中可能为当前对象添加新的引用。因此在 `finalize` 方法运行完成之后，垃圾回收器需要重新检查该对象的引用。如果发现新的引用，那么对象会回到可达状态，相当于该对象被复活；否则对象会变成不可达状态。当对象从可复活状态变为可达状态之后，对象会再次出现没有引用存在的情况。在这个情况下，`finalize` 方法不会被再次调用，对象会直接变成不可达状态，也就是说，一个对象的 `finalize` 方法只会被调用一次。第三种是不可达状态，在这个状态下，垃圾回收器可以自由地释放对象所占用的内存空间。

3.4.4 引用队列

引用队列的主要作用是作为一个通知机制。当对象的可达状态发生改变时，如果程序希望得到通知，可以使用引用队列。当从引用队列中获取了引用对象之后，不可能再获取所指向的具体对象。对于软引用和弱引用来说，在被放入队列之前，它们的引用关系就已经被清除了；而幽灵引用的 `get` 方法总是返回 null。

`WeakHashMap` 类的实现中的 `Entry` 类表示哈希表中包含的条目。`Entry` 类继承自 `WeakReference` 类，这样就可以比较方便地处理从引用队列中获取的引用对象，因为引用对象本身代表了哈希表中的条目。`Entry` 类中也包含了与条目相关的基本信息，包括键对象、值对象和引用队列的引用等。`Entry` 类作为一个弱引用，指向的是条目的键对象，而值对象仍然由强引用来指向。在程序的运行过程中，`WeakHashMap` 类的对象中的某些条目的键对象可能变成了弱引用可达的状态。垃圾回收器会清除这些弱引用并将其放入 `WeakHashMap` 类的对象的引用队列中。`WeakHashMap` 类的对象需要在合适的时机检查这个队列中包含的引用对象。由于引用对象本身就是 `Entry` 类的对象，因此

可以直接把引用对象从 `WeakHashMap` 类的对象中删除。

删除引用队列中的条目是通过 `WeakHashMap` 类中的私有方法 `expungeStaleEntries` 来完成的。在 `WeakHashMap` 类中，大部分涉及条目的方法的实现都会直接或间接地调用 `expungeStaleEntries` 方法来处理 `WeakHashMap` 类的对象中引用队列所包含的条目。这也是 `expungeStaleEntries` 方法使用 `poll` 来检查引用队列的原因。由于对 `expungeStaleEntries` 方法的调用会比较频繁，会对 `WeakHashMap` 类中的正常操作的性能产生影响，因此使用非阻塞式的 `poll` 方法是个更好的选择。

鉴于 `WeakHashMap` 类的实现机制，当其中包含的条目的键对象变成弱引用可达之后，在下次对 `WeakHashMap` 类的对象进行操作时，这些键对象对应的条目才会被删除，所以必须注意这种情况，如果对 `WeakHashMap` 类的对象的操作比较少，那么使用 `WeakHashMap` 类的对象也会出现某些键对象的存活时间过长的情况。

3.4.5 虚引用 (Phantom Reference)

“虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期，所以虚引用又被称为“幽灵引用”。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。虚引用的主要目的是在一个对象所占的内存被实际回收之前得到通知，从而可以进行一些相关的清理工作。虚引用在使用方式上与之前介绍两种引用类型有很大的不同：首先虚引用在创建时必须提供一个引用队列作为参数；其次虚引用对对象的 `get` 方法总是返回 `NULL`，因此无法通过虚引用来获取被引用的对象。

虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 (`ReferenceQueue`) 联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。

前面介绍过，软引用和弱引用在其可达状态达到时就可能被添加到对应的引用队列中。也就是说，当一个对象变成软引用可达或弱引用可达的时候，指向这个对象的引用对象就可能被添加到引用队列中。在添加到队列之前，垃圾回收器会清除掉这个引用对象的引用关系。当软引用和弱引用进入队列之后，对象的 `finalize` 方法可能还没有被调用。在 `finalize` 方法执行之后，该对象有可能重新回到可达状态。如果该对象回到了可达状态，而指向该对象的软引用或弱引用对象的引用关系已经被清除，那么就无法再通过引用对象来查找这个对象。而幽灵引用则不同，只有在对象的 `finalize` 方法被运行之后，幽灵引用才会被添加到队列中。与软引用和弱引用不同的是，幽灵引用在被添加到队列之前，垃圾回收器不会自动清除其引用关系，需要调用 `clear` 方法来显式地清除。当幽灵引用被清除之后，对象就进入了不可达状态，垃圾回收器可以回收其内存。当幽灵引用被添加到队列之后，由于 `PhantomReference` 类的 `get` 方法总是返回 `null`，程序也不能对幽灵引用所指向的对象进行任何操作。这就避免了 `finalize` 方法可能会出现对象复活的问题。幽灵引用是作为一个通知机制而存在的。程序应该在得到通知之后进行与当前对象相关的清理工作。

下面的示例给出了幽灵引用对象的使用方式。在类 `ReferencedObject` 中覆写 `finalize` 方法来提供自定义的销毁逻辑。这里只是简单地在控制台输出提示信息。在使用幽灵引用队列时，通过队列的 `poll` 方法来进行轮询。如果队列为空，就通过 `System.gc` 方法来建议垃圾回收器运行。运行示例之后会发现 `finalize` 方法中输出的消息总是最早出现的，这说明当幽灵引用进入队列之后，`finalize` 方法已经被运行过了。如果改用软引用或弱引用来进行相同试验，会发现多次运行的结果

并不一致，这是因为软引用和弱引用进入队列的时机和 `finalize` 方法的调用之间并没有必然的先后关系，如代码清单 3-113 所示。

代码清单 3-113 幽灵引用示例

```
import java.lang.ref.PhantomReference;
import java.lang.ref.Reference;
import java.lang.ref.ReferenceQueue;

public class UseReferenceQueue {
    private static class ReferencedObject{
        protected void finalize() throws Throwable{
            System.out.println("finalize 方法被调用");
            super.finalize();
        }
    }

    public void phantomReferenceQueue() {
        ReferenceQueue<ReferencedObject> queue = new ReferenceQueue<>();
        ReferencedObject obj = new ReferencedObject();
        PhantomReference<ReferencedObject> phantomRef = new PhantomReference
<ReferencedObject>(obj, queue);
        obj = null;
        Reference<? extends ReferencedObject> ref = null;
        while((ref = queue.poll()) == null){
            System.gc();
        }
        phantomRef.clear();
        System.out.println(ref == phantomRef); //值为 true
        System.out.println("幽灵引用被清除。");
    }
}
```

运行程序输出如代码清单 3-114 所示。

代码清单 3-114 3-113 运行输出

```
finalize 方法被调用
true
幽灵引用被清除。
```

如果希望在一个对象的内存被回收之前进行某些清理工作，那么相对于使用 `finalize` 方法来说，使用幽灵引用是更好的选择。幽灵引用避免了 `finalize` 方法可能造成对象复活的问题，减少了开发时可能出现的错误。不过幽灵引用的使用比 `finalize` 方法要复杂得多。最主要的问题是从引用队列中获取幽灵引用之后，无法获取其指向的对象，也就无法对这个对象进行操作。幽灵引用本身只作为一个通知机制存在，必须存在其他指向此对象的引用。因此，相对于使用幽灵引用，开发人员更倾向于谨慎使用 `finalize` 方法。只要 `finalize` 方法的实现避免了对对象复活的问题，就是一个不

错的选择。

一个比较实际的幽灵引用的应用是在虚拟机内存总量受限的情况下，可能需要等待一个占用内存空间较大的对象被回收之后再申请新的内存空间。通过这种方式，可以把程序中某部分占用的内存控制在一定的范围之内。下面的示例中，类 `PhantomAllocator` 用来分配一个字节数组供调用者使用。当每次分配新的字节数组时，会先确保之前分配的内存空间被成功释放。在每次分配新的字节数组之前，引用队列的 `remove` 方法会处于阻塞状态，直到有新的引用对象被添加到队列中。`remove` 方法返回之后，之前的字节数组的内存已经可以被释放，通过调用 `System.gc` 方法要求垃圾回收器马上回收这些内存。等内存回收之后再创建新的字节数组，创建一个幽灵引用指向新的字节数组，并与引用队列关联起来。

代码清单 3-115 申请新的内存空间示例

```
import java.lang.ref.PhantomReference;
import java.lang.ref.Reference;
import java.lang.ref.ReferenceQueue;

public class PhantomAllocator {
    private byte[] data = null;
    private ReferenceQueue<byte[]> queue = new ReferenceQueue<byte[]>();
    private Reference<? extends byte[]> ref = null;

    public byte[] get(int size){
        if(ref == null){
            data = new byte[size];
            ref = new PhantomReference<byte[]>(data,queue);
        }else{
            data = null;
            System.gc();
            try{
                ref = queue.remove();
                ref.clear();
                ref = null;
                System.gc();
                data = new byte[size];
                ref = new PhantomReference<byte[]>(data,queue);
            }catch(InterruptedException e){
                e.printStackTrace();
            }
        }
        return data;
    }
}
```

在上面的代码中，通过“`data=null`”来清除 `PhantomAllcator` 类对象本身对字节数组的强引用。在进行测试的时候，可以进行多次字节数组分配操作，同时使用工具来观察程序所占用的堆内存的情况。实际的运行结果是，程序的堆内存的占用量的峰值会维持在一个相对稳定的值。

程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

虚引用时所有引用类型中最弱的一个。一个持有虚引用的对象，和没有引用几乎一样，随时都可能被垃圾回收器回收。当试图通过虚引用的 `get()` 方法取得强引用时，总是会失败。并且，虚引用必须和引用队列一起使用，它的作用在于跟踪垃圾回收过程。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在垃圾回收后销毁这个对象时，将这个虚引用加入引用队列。

调入该句代码，`PhantomReference Ref = new PhantomReference<MyObject>(obj,softQueue);` //创建虚引用，输出结果显示对虚引用的 `get()` 操作总是返回 `null`，即便强引用还存在时也不例外。

综上所述，虚引用最大的作用在于跟踪对象回收，清理被销毁对象的相关资源。通常，当对象不被使用时，重载该类的 `finalize()` 方法可以回收对象的资源。但是如果 `finalize()` 方法使用不慎，可能导致该对象复活。一个错误的 `finalize()` 实现可能导致内存溢出，或者对象永远无法被回收，如代码清单 3-116 所示。

代码清单 3-116 虚引用错误实现

```
@Override
```

```
protected void finalize() throws Throwable{
    super.finalize();
    MyObject obj = null;
    System.out.println("MyObject's finalize called");//只有当 GC 回收时才会输出
    obj=this;
    System.out.println(obj);
}
```

上述错误需要调用两次 `obj=null`，去除该对象的强引用。由于 `finalize()` 只会被调用一次，因此，在第二次回收时，对象就没有机会再度复活了。

虚引用在使用的时候只能通过引用队列来操作。虚引用的最大优势在于引用对象被添加到队列中的时机。Java 语言提供了对象终止（`finalization`）机制来允许开发人员提供对象被销毁之前的自定义处理逻辑。Object 类提供了 `finalize` 方法来添加自定义的销毁逻辑。如果一个类有特殊的销毁逻辑，可以覆写 `finalize` 方法。从功能上来说，`finalize` 方法与 C++ 中的析构函数比较相似，但是 Java 采用的是基于垃圾回收器的自动内存管理机制，所以 `finalize` 方法在本质上不同于 C++ 中的析构函数。当垃圾回收器发现没有引用指向一个对象时，会调用这个对象的 `finalize` 方法。通常在这个方法进行一些资源释放和清理的工作，比如关闭文件、套接字和数据库连接等。由于 `finalize` 方法的存在，虚拟机中的对象一般处于三种可能的状态。第一种是可达状态，当有引用指向该对象时，该对象处于可达状态。根据引用类型的不同，有可能处于强引用可达、软引用可达或弱引用可达状态。第二种是可复活状态，如果对象的类覆写了 `finalize` 方法，则对象有可能处于该状态。虽然垃圾回收器是在对象没有引用的情况下才调用其 `finalize` 方法，但是在 `finalize` 方法的实现中可能为当前对象添加新的引用。因此在 `finalize` 方法运行完成之后，垃圾回收器需要重新检查该对象的引用。如果发现新的引用，那么对象会回到可达状态，相当于该对象被复活；否则对象会变成不可达状态。当对象从可复活状态变为可达状态之后，对象会再次出现没有引用存在的情况。在这个情况下，`finalize` 方法不会被再次调用，对象会直接变成不可达状态，也就是说，一个对象的 `finalize` 方法只会被调用一次。第三种是不可达状态，在这个状态下，垃圾回收器可以

自由地释放对象所占用的内存空间。

虚引用的用途较少，主要用于辅助 `finalize` 函数的使用。虚对象指一些对象，它们执行完了 `finalize` 函数，并为不可达对象，但是它们还没有被 GC 回收。这种对象可以辅助 `finalize` 进行一些后期的回收工作，我们通过覆盖引用的 `clear()` 方法，增强资源回收机制的灵活性。总的来说，虚引用可以用来做一些精细的内存控制操作。虚引用是专门针对引用队列的一种引用申明，例如你声明虚引用的时候是要传入一个队列，当你的虚引用所引用的对象已经执行完 `finalize` 函数的时候，就会把对象加到队列里面，你可以通过判断队列里面是不是有对象来判断你的对象是不是要被回收了。

综合上面所有引用类型的描述，根据 GC 的工作原理，我们可以通过一些技巧和方式，让 GC 运行更加有效率，更加符合应用程序的要求。以下是一些程序设计的几点建议。

- (1) 最基本的建议就是尽早释放无用对象的引用。大多数程序员在使用临时变量的时候，都是让引用变量在退出活动域(scope)后，自动设置为 `null`。我们在使用这种方式时候，必须特别注意一些复杂的对象图，例如数组，队列，树，图等，这些对象之间有相互引用关系较为复杂。对于这类对象，GC 回收它们一般效率较低。如果程序允许，尽早将不用的引用对象赋为 `null`。这样可以加速 GC 的工作。
- (2) 尽量少用 `finalize` 函数。`finalize` 函数是 Java 提供给程序员一个释放对象或资源的机会。但是，它会加大 GC 的工作量，因此尽量少采用 `finalize` 方式回收资源。
- (3) 如果需要使用经常使用的图片，可以使用 `soft` 应用类型。它可以尽可能将图片保存在内存中，供程序调用，而不引起 `OutOfMemory`。
- (4) 注意集合数据类型，包括数组、树、图、链表等数据结构，这些数据结构对 GC 来说，回收更为复杂。另外，注意一些全局的变量，以及一些静态变量。这些变量往往容易引起悬挂对象(dangling reference)，造成内存浪费。
- (5) 当程序有一定的等待时间，程序员可以手动执行 `System.gc()`，通知 GC 运行，但是 Java 语言规范并不保证 GC 一定会执行。使用增量式 GC 可以缩短 Java 程序的暂停时间。

3.5 其他相关概念

3.5.1 JNI 技术提升

JavaCPP 是一个开源库，它提供了在 Java 中高效访问本地 C++ 的方法。采用 JNI 技术实现，所以支持所有 Java 实现包括 Android⁹ 系统，Avian¹⁰ 和 RoboVM¹¹。

⁹ 一种基于 Linux 的自由及开放源代码的操作系统，主要使用于移动设备，如智能手机和平板电脑，由 Google 公司和开放手机联盟领导及开发。

¹⁰ Avian 是一个轻量级的 Java 虚拟机和类库，提供了 Java 特性的一个有用的子集，适合开发跨平台、自包容的应用程序。

¹¹ RoboVM 编译器可以将 Java 字节码翻译成 ARM 或者 x86 平台上的原生代码，应用可直接在 CPU 上运行，无须其他解释器或者虚拟机。RoboVM 同时包含一个 Java 到 Objective-C 的桥，可像其他 Java 对象一样来使用 Objective-C 对象。大多数 UIKit 已经支持，而且将会支持更多的框架。

总的来说，JavaCPP 提供了一系列的 Annotation 将 Java 代码映射到 C++代码，并使用一个可执行的 JAR 包将 C++代码转化为可以从 JVM 内调用的动态链接库文件。

与其他技术相比，比较特性总结如表 3-1 所示。

表 3-1 类似技术介绍或特点

技术名称	技术介绍
CableSwig	用于针对 Tcl 和 Python 语言创建接口
JNIGeneratorApp	所有用于 SWT 的 C 代码都是通过它来创建的
cxxwrap	用于生成针对 C++的 Java JNI 包、HTML 文档、用户手册
JNIWrapper	商业版本，可以帮助实现 Java 和本地代码之间的无缝结合
Platform Invoke	微软发布的一个工具
GlueGen	针对 C 语言的一个工具，帮助生成 JNI 代码
LWJGL Generator	JNI 代码生成器
ctypes	针对 Python 的接口代码生成器
JNA	JNA（Java Native Access）提供一组 Java 工具类用于在运行期动态访问系统本地库（native library：如 Window 的 dll）而不需要编写任何 Native/JNI 代码。开发人员只要在一个 java 接口中描述目标 native library 的函数与结构，JNA 将自动实现 Java 接口到 native function 的映射
JNIEasy	替换 JNA 的一种技术
JNative	Windows 版本的库（DLL），提供了 JNI 代码生成
ffixxx	针对 haskell 模型的代码生成器，主要生成 C 语言
JavaCPP	更加自然高效，它支持大部分的 C++语法特性。目前已经能成功封装 OpenCV, FFmpeg, libdc1394, PGR FlyCapture, OpenKinect, videoInput, and ARToolKitPlus。除此之外，它还能直接把 C/C++的头文件转化成 Java 类，能自动生成 JNI 代码，编译成本地库，开发人员无须编写烦琐的 C++，JNI 代码，从而提高开发效率

为了调用本地方法，JavaCPP 生成了对应的 JNI 代码，并且把这些代码输入到 C++编译器，用来构建本地库。使用了 Annotations特性的 Java 代码在运行时会自动调用 Loader.load()方法从 Java 资源里载入本地库，这里指的资源是工程构建过程中配置好的。

我们先来演示一个例子，这是一个简单的注入/读出方法，类似于 JavaBean 的工作方式。代码 3-117 所示的 LegacyLibrary.h 包含了 C++类。

代码清单 3-117 LegacyLibrary.h

```
#include <string>

namespace LegacyLibrary {

    class LegacyClass {
    public:
        const std::string& get_property() { return property; }
        void set_property(const std::string& property) { this->property =
property; }
        std::string property;
    };
}
```


接下来定义一个 Java 类，驱动 JavaCPP 来完成调用 C++ 代码，如代码清单 3-118 所示。

代码清单 3-118 LegacyLibrary.java

```
import org.bytedeco.javacpp.*;
import org.bytedeco.javacpp.annotation.*;
@Platform(include="LegacyLibrary.h")
@Namespace("LegacyLibrary")
public class LegacyLibrary {
    public static class LegacyClass extends Pointer {
        static { Loader.load(); }
        public LegacyClass() { allocate(); }
        private native void allocate();

        // to call the getter and setter functions
        public native @StdString String get_property(); public native void
set_property(String property);

        // to access the member variable directly
        public native @StdString String property(); public native void property
(String property);
    }

    public static void main(String[] args) {
        // Pointer objects allocated in Java get deallocated once they become
unreachable,
        // but C++ destructors can still be called in a timely fashion with
Pointer.deallocate()
        LegacyClass l = new LegacyClass();
        l.set_property("Hello World!");
        System.out.println(l.property());
    }
}
```

以上两个类放在一个目录下面，接下来运行一系列编译指令，如代码清单 3-119 所示。

代码清单 3-119 运行命令

```
$ javac -cp javacpp.jar LegacyLibrary.java
$ java -jar javacpp.jar LegacyLibrary
$ java -cp javacpp.jar LegacyLibrary
Hello World!
```

我们看到代码清单 3-119 最后以行输出了一行“Hello World!”，这是 LegacyLibrary 类里面定义好的，通过一个 setter 方法注入字符串，getter 方法读出字符串。

我们可以看到文件夹里面内容的变化，刚开始的时候只有.h、.java 两个文件，代码清单 3-119 所示的 3 个命令运行过后，生成了 class 文件及本地方法(native method)对应的.so 文件，如代码清单 3-120 所示。

方式的效率则平均在 0.81 左右。

表 3-3 JNI 库和 C++库多线程性能比较

方式	线程数	比对次数 (万次)	总耗时 /ms	总比对速率/rps (records per second)	单线程效 率/rps	CPU/ 百分比	C++/JNI 效率系数
C++	5	500	6313	396W	87W	500	1
	10	500	9477	528w	71W	1000	1
	15	500	12496	600w	50W	1500	1
	20	500	15581	642w	36W	1600	1
	25	500	19257	649w	27W	1600	1
	50	500	37512	666w	14W	1600	1
	100	500	74773	675w	7.0W	1600	1
JNI 方式	5	500	7907	316W	68W	500	0.80
	10	500	11700	427W	57W	1000	0.81
	15	500	14846	505W	37W	1500	0.84
	20	500	19541	512W	28W	1600	0.80
	25	500	24488	521W	26W	1600	0.80
	50	500	46629	536W	13W	1600	0.80
	100	500	91939	544W	5.8W	1600	0.81
JavaCPP 方式	5	500	6753	370W	83W	500	0.93
	10	500	8440	592W	63W	1000	1.12
	15	500	12680	591W	40W	1500	0.99
	20	500	17390	575W	29W	1600	0.89
	25	500	21939	570W	22W	1600	0.87
	50	500	43849	570W	11W	1600	0.85
	100	500	83150	601W	5.7W	1600	0.89

3.5.2 异常捕获机制

异常处理以一种简洁的方式表示了程序中可能出现的错误，以及应对这些错误的处理方式。适当地使用异常处理技术，可以提高代码的可靠性、可维护性和可读性。但是如果使用不当，就会产生相反的效果。比如虽然一个方法声明了会抛出某个异常，但是使用这个方法的代码在异常发生的时候，却只能捕获异常之后就直接忽略它，无法做其他的处理。而为了能够通过编译，又不得不加上 catch 语句。这势必会造成冗余无用的代码，同时给出不适当的异常设计的一个信号。

Java 7 对异常处理做了两个重要的改动：一个是支持在一个 catch 子句中同时捕获多个异常，另外一个是在捕获并重新抛出异常时的异常类型更加精确。

Java 语言中基本的异常处理是围绕 try-catch-finally、throws 和 throw 这几个关键词展开的。具体来说，throws 用来声明一个方法可能抛出的异常，对方法体中可能抛出的异常都要进行声明；throw 用来在遇到错误的时候抛出一个具体的异常；try-catch-finally 则用来捕获异常并进行处理。Java 中的异常有受检异常和非受检异常两类。

1. 受检异常和非受检异常

在异常处理的时候，都会接触到受检异常（checked exception）和非受检异常（unchecked

exception) 这两种异常类型。非受检异常指的是 `java.lang.RuntimeException` 和 `java.lang.Error` 类及其子类, 所以其他的异常类都称为受检异常。两种类型的异常在作用上并没有差别, 唯一的差别就在于使用受检异常时的合法性要在编译时刻由编译器来检查。正因为如此, 受检异常在使用的时候需要比非受检异常更多的代码来避免编译错误。

受检异常的特点在于它强制要求开发人员在代码中进行显式的声明和捕获, 否则就会产生编译错误。这种限制从好的方面来说, 可以防止开发人员意外地忽略某些出错的情况, 因为编译器不允许出现未被处理的受检异常; 从不好的方面来说, 受检异常对程序中的设计提出了更高的要求。不恰当地使用受检异常, 会使代码中充斥着大量没有实际作用、只是为了通过编译而添加的代码。而非受检异常的特点是, 如果不捕获异常, 不会产生编译错误, 异常会在运行时刻才被抛出。非受检异常的好处是可以去掉一些不需要的异常处理代码, 而不好之处是开发人员可能忽略某些应该处理的异常。一个典型的例子是把字符串转换成数字时会发生 `java.lang.NumberFormatException` 异常, 忽略该异常可能导致一个错误的输入就造成整个程序退出。

目前的主流意见是最好优先使用非受检异常。

2. 异常声明是 API 的一部分

这一条提示主要是针对受检异常的。在一个公开方法的声明中使用 `throws` 关键词来声明其可能抛出的异常的时候, 这些异常就成为这个公开方法的一部分, 属于开放 API。在维护这个公开 API 的时候, 这些异常有可能会对 API 的演化造成障碍, 使得编写代码时不得不考虑向后兼容性的问题。

如果公开方法声明了会抛出一个受检异常, 那么这个 API 的使用者肯定已经使用了 `try-catch-finally` 来处理这个异常。如果在后面的版本更新中, 发现该 API 抛出这个异常是不合适的, 也不能直接把这个异常的声明删除。因为这样会造成之前的 API 使用者的代码无法通过编译。

因此, 对于 API 的设计者来说, 谨慎考虑每个公开方法所声明的异常是很有必要的。因为一旦加了异常声明, 在很长的一段时间内都无法甩开它。这也是为什么推荐使用非受检异常的一个重要原因, 非受检异常不需要声明就可以直接抛出。但是对于一个方法会抛出的非受检异常, 也需要在文档中进行说明。

1. 精心设计异常的层次结构

一般来说, 一个程序中应该要有自己的异常类的层次结构。如果只打算使用非受检异常, 至少需要一个继承自 `RuntimeException` 的异常类。如果还需要使用受检异常, 还要有另外一个继承自 `Exception` 的异常类。如果程序中可能出现的异常情况比较多, 应该在不同的抽象层次上定义相关的异常, 并形成完整的层次结构。这个异常的层次结构与程序本身的类层次结构是相对应的。不同抽象层次上的代码应该只声明抛出同一层次的相关异常。

比如一个典型的 Web 应用按照自顶向下的顺序一般分成展现层、服务层和数据访问层。与之对应的异常也应该按照这个层次结构来进行划分。数据访问层的代码应该只声明抛出与访问数据相关的异常, 如数据库连接和操作相关的异常。这么做的好处是工作于某个抽象层次上的开发人员不需要去了解其他层次上的细节。比如服务层开发人员会调用数据抽象层的代码, 他只需要关心数据访问可能出现异常即可, 而并不需要去关心这是一个数据库访问异常, 还是一个文件系统访问异常。这种抽象层次的划分对系统的演化是比较重要的。假如系统以后不再使用数据库作为

数据访问的实现，服务层的异常处理逻辑也不会受到影响。

一般来说，对于程序中可能出现的各种错误，都需要声明一个异常类与之对应。有些开发人员会选择一个大而全的异常类来表示各种不同类型的错误，利用这个异常的消息来区分不同的错误。比如声明一个异常类 `BaseException`，不管是数据访问错误还是用户输入的数据格式不对，都会抛出同一个异常，只是使用的消息内容不同。当采用这种异常设计方式的时候，异常的处理者只能根据异常消息字符串的内容来判断具体的错误类型。如果异常的处理者只是简单地进行日志记录或重新抛出此异常，这种方式并没有太大的问题。如果异常的处理者需要解析异常的消息格式来判断具体类型，那么这种方式就是不可取的，应该换成不同的异常类。

采用这种异常层次结构会遇到一个常见的异常处理模式是包装异常。包装异常的目的在于使异常只出现在其所对应的抽象层次上。当一个异常抛出的时候，如果没有被捕获到，就会一直沿着调用栈往上传递，直到被上层方法捕获或是最终由 Java 虚拟机来处理。这种传递方式会使这个异常跨越多个抽象层次的边界，使得上层代码看到不需要关注的底层异常。为此，在一个异常要跨越抽象层次边界的时候，需要进行包装。包装之后的异常才是上层代码需要关注的。

对一个异常进行包装是一件非常简单的事情。从 JDK1.4 开始，所有异常的基类 `java.lang.Throwable` 就支持在构造方法中传入另外一个异常作为参数。而这个参数所表示的异常被包装在新的异常中，可以通过 `getCause` 方法来获取。下面代码给出了一个异常包装的示例，把底层的 `IOException` 包装成更为抽象的 `DataAccessException`。使用 `DataAccessGateway` 类的上层代码只需要知道 `DataAccessException` 即可，并不需要知道 `IOException` 的存在。

代码清单 3-121 异常包装示例

```
import java.io.FileInputStream;

public class DataAccessGateway {

    public void load() throws DataAccessException{
        try{
            FileInputStream input = new FileInputStream("data.txt");
        }catch(IOException e){
            throw new DataAccessException(e);
        }
    }
}
```

在使用异常包装的时候，一个典型的做法就是为每个层次定义一个基本的异常类。这个层次的所有公开方法在声明异常的时候都是用这个异常类。所有这个层次中出现的底层异常都被包装成这个异常。

2. 异常类中包含足够的信息

异常存在的一个很重要的意义在于，当错误发生的时候，调用者可以对错误进行处理，从产生的错误中恢复。为了方便调用者处理这些异常，每个异常中都需要包含尽量丰富的信息。异常不应该只说明某个错误发生了，还应该给出相关的信息。异常类是完整的 Java 类，因此在其中添加所需的域和方法是一件很简单的事情。

考虑这样一个场景，当用户进行支付的时候，如果他的当前余额不足以完成支付，那么在所

抛出的异常信息中，可以包含当前所需的金额、余额和其中的差额等信息。这样异常处理者就可以提供给用户更加具体的出错信息。

Java7 的异常处理特性总结如下。

1. 一个 catch 子句捕获多个异常

在 Java7 之前的异常处理语法中，一个 catch 子句只能捕获一类异常。在要处理的异常种类很多时这种限制会很麻烦。每一种异常都需要添加一个 catch 子句，而且这些 catch 子句中的处理逻辑可能都是相同的，从而会造成代码重复。虽然可以在 catch 子句中通过这些异常的基类来捕获所有的异常，比如使用 Exception 作为捕获的类型，但是这要求对这些不同的异常所做的处理是相同的。另外也可能会捕获到某些不应该被捕获的非受检异常。而在某些情况下，代码重复是不可避免的。比如某个方法可能抛出 4 种不同的异常，其中有 2 种异常使用相同的处理方式，另外 2 种异常的处理方式也相同，但是不同于前面的 2 种异常。这势必会在 catch 子句中包含重复的代码。

对于这种情况，Java7 改进了 catch 子句的语法，允许在其中指定多种异常，每个异常类型之间使用“|”来分隔，如代码清单 3-122 所示。

代码清单 3-122 Java7 改进 catch 子句方法

```
public class ExceptionHandler {
    public void handle() {
        ExceptionThrower thrower = new ExceptionThrower();
        try {
            thrower.manyExceptions();
        } catch (ExceptionA | ExceptionB ab) {
            // 处理 A 和 B 异常
        } catch (ExceptionC c) {
            // 处理 C 异常
        }
    }
}
```

ExceptionThrower 类的 manyExceptions 方法抛出 ExceptionA、ExceptionB 和 ExceptionC 三种异常，其中对 ExceptionA、ExceptionB 采用一种处理方式，对 ExceptionC 采用另外一种处理方式。

需要注意的是，catch 子句中声明捕获的这些异常类中，不能出现重复的类型，也不允许其中的某个异常时另外一个异常的子类，否则会出现编译错误。如果在 catch 子句中声明了多个异常类，那么异常参数的具体类型是所有这些异常类型的最小边界。

2. 更加精确的异常抛出

在进行异常处理的时候，如果遇到当前代码无法处理的异常，应该把异常重新抛出，交由调用栈的上层代码来处理。在重新抛出异常的时候，需要判断异常的类型。Java7 对重新抛出异常时的异常类型做了更加精确的判断，以保证抛出的异常的确是**可以被抛出的**。这个改进看起来让人有点费解，因为从语义上来说，不能被抛出来的异常是不会被重新抛出的。但是在 Java7 之前，Java 编译器并不能做出精确的判断，因此会存在一些隐含的不正确的情况。在 Java7 中，如果一个 catch 子句的异常类型参数在 catch 代码块中没有被修改，而这个异常又被重新抛出，编译器会知道这个

重新被抛出的异常肯定是 try 语句块中可以抛出的异常，同时也是没有被之前的 catch 子句捕获的异常。下面的例子说明了 Java7 之前的编译器和 Java7 编译器不一样的行为。

代码清单 3-123 Java 编译器变化

```
public class PrecideThrowUse {
    public void testThrow() throws ExceptionA{
        try{
            throw new ExceptionASub2();
        }catch(Exception e){
            try{
                throw e;
            }catch(ExceptionASub1 e2){
                //这里会抛出编译错误提示
            }
        }
    }
}
```

在上面的代码中，异常类 ExceptionASub1 和 ExceptionASub2 都是 ExceptionA 的子类，而且这两者之间并没有继承关系。方法 testThrow 中首先抛出 ExceptionASub2 异常，通过第一个 catch 子句捕获之后重新抛出。在这里，Java 编译器可以精确知道变量 e 表示的异常类型是 ExceptionASub2，接下来的第二个 catch 子句试图捕获 ExceptionASub1 类型的异常，这显然是不可能的，因此会产生编译错误。上面的代码在 Java6 编译器上是可以通过编译的。对于 Java6 编译器来说，第二个 try 子句中抛出的异常类型是前一个 catch 子句中声明的 ExceptionA 类型，因此在第二个 catch 子句中尝试捕获 ExceptionA 的子类型 ExceptionASub1 是合法的。

3.5.3 ExceptionUtils 类

org.apache.commons.lang3.exception.ExceptionUtils 类封装了 Exception 类，它们的输出结果完全一致，我们可以通过使用这个类来规避对于底层 Exception 类的理解。如代码清单 3-124 所示。

代码清单 3-124 ExceptionUtils 类使用示例

```
import java.io.File;
import java.net.URL;
import java.net.URLDecoder;

import org.apache.commons.lang3.exception.ExceptionUtils;

public class testStr {
    public String getProjectPath() throws Exception{
        String filePath = "";
        try {
            URL url = testStr.class.getProtectionDomain().getCodeSource().getLocation();
            filePath = URLDecoder.decode(url.getPath(), "utf-8");
            if (filePath.endsWith(".jar")) {
                filePath = filePath.substring(0, filePath.lastIndexOf("/"));
            }
        }
    }
}
```



```

    }
    File file = new File(filePath);
    filePath = file.getParent();
    throw new Exception();
} catch (Exception e) {
    System.out.println(ExceptionUtils.getStackTrace(e));
} catch (Exception e) {
    e.printStackTrace();
}
}
return filePath;
}

public static void main(String[] args) throws Exception{
    testStr str = new testStr();
    str.getProjectPath();
}
}

```

程序运行后输出如代码清单 3-125 所示。

代码清单 3-125 3-124 运行输出

```

java.lang.Exception
    at testStr.getProjectPath(testStr.java:18)
    at testStr.main(testStr.java:27)

```

org.apache.commons.lang.exception.ExceptionUtils.getStackTrace(e)的源代码如代码清单 3-126 所示。

代码清单 3-126 getStackTrace 源代码

```

public static String getStackTrace(Throwable throwable) {
    StringWriter sw = new StringWriter();
    PrintWriter pw = new PrintWriter(sw, true);
    throwable.printStackTrace(pw);
    return sw.getBuffer().toString();
}

```

从上面的运行输出我们可以看出，异常返回的是 String，直接调用 Exception 返回一个 StackTraceElement[]。

3.5.4 循环技巧

由于方法的同步需要消耗相当多的计算资源，所以建议不要在循环中调用 synchronized(同步)方法，关于 synchronized 方法会在第 5 章有较深入的解释。如代码清单 3-127 所示，我们调用了同步锁。

代码清单 3-127 同步锁示例

```

import java.util.Vector;

public class syn {

```

```

public synchronized void method (object o) {
}
private void test () {
    for (int i = 0; i < vector.size(); i++) {
        method (vector.elementAt(i));    // violation
    }
}
private vector vector = new vector (5, 5);
}

```

如上面代码所示，此处 `method` 方法加入了同步锁，尽量不要在循环体中调用同步方法，如果必须同步的话，推荐以下方式，如代码清单 3-128 所示。

代码清单 3-128 同步锁改进方案

```

import java.util.vector;
public class syn {
    public void method (object o) {
    }
    private void test () {
        synchronized{//在一个同步块中执行非同步方法
            for (int i = 0; i < vector.size(); i++) {
                method (vector.elementAt(i));
            }
        }
    }
    private vector vector = new vector (5, 5);
}

```

嵌套 `for` 循环中次数多的放在内侧，次数少的放在外侧。众所周知 `for` 循环需要定义一个循环变量来遍历每一个需要循环的对象，那么如果循环次数多的循环放在外侧那么无疑将会使得总体的变量增多，效率自然会降低，如代码清单 3-129 所示。

代码清单 3-129 循环方法

```

public class testFor {
    public static void main(String[] args){
        long start = System.currentTimeMillis();
        int count = 0;
        for (int i=0; i<100;i++){
            for(int j=0; j<10000000;j++){
                count++;
            }
        }
        System.out.println(count);
        System.out.println(System.currentTimeMillis() - start);
        start = System.currentTimeMillis();
        count = 0;
        for (int i=0; i<1000000;i++){

```



```

        for(int j=0; j<100;j++){
            count++;
        }
    }
    System.out.println(count);
    System.out.println(System.currentTimeMillis()- start);
}
}

```

代码 3-129 运行输出如代码清单 3-130 所示。

代码清单 3-130 3-129 运行输出

```

1000000000
1092
1000000000
172

```

3.5.5 替换 switch

关键字 switch 语句用于多条件判断，switch 语句的功能类似于 if-else 语句，两者的性能差不多。但是 switch 语句有性能提升空间，如代码清单 3-131 所示。

代码清单 3-131 Switch 提升示例

```

public class switchCompareIf {
    public static int switchTest(int value){
        int i = value%10+1;
        switch(i){
            case 1:return 10;
            case 2:return 11;
            case 3:return 12;
            case 4:return 13;
            case 5:return 14;
            case 6:return 15;
            case 7:return 16;
            case 8:return 17;
            case 9:return 18;
            default:return -1;
        }
    }

    public static int arrayTest(int[] value,int key){
        int i = key%10+1;
        if(i>9 || i<1){
            return -1;
        }else{
            return value[i];
        }
    }
}

```

```

    }

    public static void main(String[] args){
        int chk = 0;
        long start=System.currentTimeMillis();
        for(int i=0;i<10000000;i++){
            chk = switchTest(i);
        }
        System.out.println(System.currentTimeMillis()-start);
        chk = 0;
        start=System.currentTimeMillis();
        int[] value=new int[]{0,10,11,12,13,14,15,16,17,18};
        for(int i=0;i<10000000;i++){
            chk = arrayTest(value,i);
        }
        System.out.println(System.currentTimeMillis()-start);
    }
}

```

使用一个连续的数组代替 switch 语句，由于对数据的随机访问非常快，至少好于 switch 的分支判断，从上面例子可以看到比较的效率差距是近乎 1 倍，switch 方法耗时 172ms，if-else 方法耗时 93ms。

3.5.6 优化循环

当性能问题成为系统的主要矛盾时，可以尝试优化循环，例如减少循环次数，这样可以加快程序运行速度，如代码清单 3-132 所示。

代码清单 3-132 减少 Loop 循环

```

public class reduceLoop {
    public static void beforeTuning(){
        long start = System.currentTimeMillis();
        int[] array = new int[9999999];
        for(int i=0;i<9999999;i++){
            array[i] = i;
        }
        System.out.println(System.currentTimeMillis() - start);
    }

    public static void afterTuning(){
        long start = System.currentTimeMillis();
        int[] array = new int[9999999];
        for(int i=0;i<9999999;i+=3){
            array[i] = i;
            array[i+1] = i+1;
            array[i+2] = i+2;
        }
        System.out.println(System.currentTimeMillis() - start);
    }
}

```



```

    }

    public static void main(String[] args){
        reduceLoop.beforeTuning();
        reduceLoop.afterTuning();
    }
}

```

按照运行输出的时间来看，这个例子可以看出，通过减少循环次数，耗时缩短为原来的 1/8。

在循环体中实例化临时变量将会增加内存消耗，如代码清单 3-133 所示。

代码清单 3-133 循环体内实例化变量

```

import java.util.Vector;
public class loop {
    void method (Vector v) {
        for (int i=0;i < v.size();i++) {
            Object o = new Object();
            o = v.elementAt(i);
        }
    }
}

```

在循环体外定义变量，并反复使用，3-133 代码优化成 3-134 所示代码。

代码清单 3-134 循环体外方式

```

import java.util.Vector;
public class loop {
    void method (Vector v) {
        Object o;
        for (int i=0;i<v.size();i++) {
            o = v.elementAt(i);
        }
    }
}

```

3.5.7 使用 arrayCopy()

数据复制是一项使用频率很高的功能，JDK 中提供了一个高效的 API 来实现它。System.arraycopy()函数是 native 函数，通常 native 函数的性能要优于普通的函数，所以，仅处于性能考虑，在软件开发中，应尽可能调用 native 函数。

ArrayList 和 Vector 大量使用了 System.arraycopy 来操作数据，特别是同一数组内元素的移动及不同数组之间元素的复制。

arraycopy 的本质是让处理器利用一条指令处理一个数组中的多条记录，有点像汇编语言里面的串操作指令（LODSB、LODSW、LODSB、STOSB、STOSW、STOSB），只需指定头指针，然后开始循环即可，即执行一次指令，指针就后移一个位置，操作多少数据就循环多少次。

如果在应用程序中需要进行数组复制，应该使用这个函数，而不是自己实现，arrayCopy 使用示例如代码清单 3-135 所示。

代码清单 3-135 arrayCopy 使用示例

```
public class arrayCopyTest {
    public static void arrayCopy(){
        int size = 10000000;
        int[] array = new int[size];
        int[] arraydestination = new int[size];
        for(int i=0;i<array.length;i++){
            array[i] = i;
        }
        long start = System.currentTimeMillis();
        for(int j=0;j>1000;j++){
            System.arraycopy(array, 0, arraydestination, 0, size);//使用 System 级别
            的本地 arraycopy 方式
        }
        System.out.println(System.currentTimeMillis() - start);
    }

    public static void arrayCopySelf(){
        int size = 10000000;
        int[] array = new int[size];
        int[] arraydestination = new int[size];
        for(int i=0;i<array.length;i++){
            array[i] = i;
        }
        long start = System.currentTimeMillis();
        for(int i=0;i<1000;i++){
            for(int j=0;j<size;j++){
                arraydestination[j] = array[j];//自己实现的方式，采用数组的数据互换方式
            }
        }
        System.out.println(System.currentTimeMillis() - start);
    }

    public static void main(String[] args){
        arrayCopyTest.arrayCopy();
        arrayCopyTest.arrayCopySelf();
    }
}
```

代码 3-135 运行输出如清单 3-136 所示。

代码清单 3-136 3-135 运行输出

0 23166

3.5.8 使用 Buffer 进行 I/O 操作

除 NIO 外,使用 Java 进行 I/O 操作有两种基本方式,一是使用基于 `InputStream` 和 `OutputStream` 的方式,二是使用 `Writer` 和 `Reader`。

无论使用哪种方式进行文件 I/O,如果能合理地使用缓冲,就能有效地提高 I/O 的性能。

下面显示了可与 `InputStream`、`OutputStream`、`Writer` 和 `Reader` 配套使用的缓冲组件。

`OutputStream-FileOutputStream-BufferedOutputStream`

`InputStream-FileInputStream-BufferedInputStream`

`Writer-FileWriter-BufferedWriter`

`Reader-FileReader-BufferedReader`

使用缓冲组件对文件 I/O 进行包装,可以有效提高文件 I/O 的性能,如代码清单 3-137 所示。

代码清单 3-137 使用缓冲组件

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class StreamVSBuffer {
    public static void streamMethod() throws IOException{
        try {
            long start = System.currentTimeMillis();
            DataOutputStream dos = new DataOutputStream(new FileOutputStream(
"C:\\StreamVSBufferTest.txt")); //请替换成自己的文件
            for(int i=0;i<10000;i++){
                dos.writeBytes(String.valueOf(i)+"\\r\\n");//循环 1 万次写入数据
            }
            dos.close();
            DataInputStream dis = new DataInputStream(new FileInputStream("C:\\
StreamVSBufferTest.txt"));
            while(dis.readLine() != null){

            }
            dis.close();
            System.out.println(System.currentTimeMillis() - start);
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

```

    public static void bufferMethod() throws IOException{
        try {
            long start = System.currentTimeMillis();
            DataOutputStream dos = new DataOutputStream(new BufferedOutputStream(
            new FileOutputStream("C:\\StreamVSBuffertest.txt")));//请替换成自己的文件
            for(int i=0;i<10000;i++){
                dos.writeBytes(String.valueOf(i)+"\r\n");//循环 1 万次写入数据
            }
            dos.close();
            DataInputStream dis = new DataInputStream(new BufferedInputStream(new
            FileInputStream("C:\\StreamVSBuffertest.txt")));
            while(dis.readLine() != null){
                // TODO Auto-generated catch block
            }
            dis.close();
            System.out.println(System.currentTimeMillis() - start);
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    public static void main(String[] args){
        try {
            StreamVSBuffer.streamMethod();
            StreamVSBuffer.bufferMethod();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

从代码清单 3-137 的输出（分别为 889 和 31 毫秒）我们可以看到，很明显使用缓冲的代码性能比没有使用缓冲的快了很多倍。

下面的代码对 `FileWriter` 和 `FileReader` 进行了相似的测试，如代码清单 3-138 所示。

代码清单 3-138 `FileWriterVSBFileReader`

```

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class WriterVSBuffer {

```



```

public static void streamMethod() throws IOException{
    try {
        long start = System.currentTimeMillis();
        FileWriter fw = new FileWriter("C:\\StreamVSBuffertest.txt");//请替换
成自己的文件
        for(int i=0;i<10000;i++){
            fw.write(String.valueOf(i)+"\r\n");//循环 1 万次写入数据
        }
        fw.close();
        FileReader fr = new FileReader("C:\\StreamVSBuffertest.txt");
        while(fr.ready() != false){
        }
        fr.close();
        System.out.println(System.currentTimeMillis() - start);
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

public static void bufferMethod() throws IOException{
    try {
        long start = System.currentTimeMillis();
        BufferedWriter fw = new BufferedWriter(new FileWriter("C:\\
StreamVSBuffertest.txt"));//请替换成自己的文件
        for(int i=0;i<10000;i++){
            fw.write(String.valueOf(i)+"\r\n");//循环 1 万次写入数据
        }
        fw.close();
        BufferedReader fr = new BufferedReader(new FileReader("C:\\
StreamVSBuffertest.txt"));
        while(fr.ready() != false){
        }
        fr.close();
        System.out.println(System.currentTimeMillis() - start);
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

public static void main(String[] args){
    try {
        StreamVSBuffer.streamMethod();
        StreamVSBuffer.bufferMethod();
    }
}

```

```

    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

```

从上面例子的输出（分别为 1295 和 31 毫秒）可以看出，无论对于读取还是写入文件，适当地使用缓冲，可以有效地提升系统的文件读写性能，为用户减少响应时间。

3.5.9 使用 clone() 代替 new

在 Java 中新建对象实例最常用的方法是使用 new 关键字。JDK 对 new 关键字的支持非常好，使用 new 关键字创建轻量级对象时，速度非常快。但是，对于重量级对象，由于对象在构造函数中可能会进行一些复杂且耗时的操作，因此，构造函数的执行时间可能会比较长。这就导致创建对象的耗时很长，同时也使得系统无法在短期内获得大量的实例。为了解决这个问题，可以使用 Object.clone() 方法。Object.clone() 方法可以绕过对象构造函数，快速复制一个对象实例。由于不需要调用对象构造函数，因此，clone() 方法不会受到构造函数性能的影响，能够快速生成一个实例。但是，在默认情况下，clone() 方法生成的实例只是原对象的浅拷贝。如果需要深拷贝，则需要重新实现 clone() 方法。

下面这个例子是一个实现了 Cloneable 接口的 JavaBean，它拥有一个通过 clone() 方法生成新实例的 newInstance() 函数。此外，它的构造函数性能较差，通过构造函数生成对象的效率很低。

代码清单 3-139 Cloneable 接口实现

```

import java.util.Vector;
import org.apache.hadoop.hbase.util.Threads;

public class cloneDemo implements Cloneable{
    private int id;
    private String str;
    private Vector v;
    public cloneDemo() throws InterruptedException{
        Thread.sleep(1000);
        System.out.println("Constructor called");
    }

    public int getId(){
        return id;
    }

    public void setId(int id){
        this.id = id;
    }

    public String getStr(){
        return str;
    }
}

```



```

    }

    public void setStr(String str){
        this.str = str;
    }

    public Vector getV(){
        return v;
    }

    public void setV(Vector v){
        this.v = v;
    }

    public cloneDemo newInstance(){
        try{
            return (cloneDemo)this.clone();//使用 clone 方法创建对象, 属于浅拷贝
        }catch(CloneNotSupportedException ex){
            ex.printStackTrace();
        }
        return null;
    }

```

```

    public static void main(String[] args) throws InterruptedException{
        long start = System.currentTimeMillis();
        cloneDemo c1 = new cloneDemo();
        System.out.println(System.currentTimeMillis() - start);
        Vector v = new Vector();
        v.add("java");
        c1.setId(1);
        c1.setStr("test1");
        c1.setV(v);

        start = System.currentTimeMillis();
        cloneDemo c2 = c1.newInstance();
        System.out.println(System.currentTimeMillis() - start);
        c2.setId(2);
        c2.setStr("test2");
        c2.getV().add("C++");

        System.out.println("c1==c2?" + (c1==c2));
        System.out.println(c1.getV()==c2.getV());
        System.out.println(c1.getStr());
        System.out.println(c1.getId());
        System.out.println(c2.getStr());
        System.out.println(c2.getId());
    }

```

代码 3-139 运行输出如清单 3-140 所示。

代码清单 3-140 3-139 运行输出

```
Constructor called
1014
0
c1==c2?false
true
test1
1
test2
2
```

如果需要实现深拷贝,则需要重载 `Object` 对象的 `clone()` 方法。首先使用 `super.clone()` 方法生成一份浅拷贝对象。然后,生成一个新的 `Vector` 实例,将原实例内容复制到新的实例中,使克隆后的对象与原对象持有不同的引用,实现了简单的深拷贝。

3.5.10 I/O 速度

对于 CPU 密集型的程序,即程序中包含大量计算,Java 程序可以达到 C/C++ 程序同等级别的速度,但是对于 I/O 密集型的程序,即程序中包含大量 I/O 操作,Java 程序的速度就远远慢于 C/C++ 程序了,很大程度上是因为 C/C++ 程序能直接访问底层的存储设备。

在 Java I/O 中最基本的概念是流 (`Stream`),流是一个连续的字节序列,包括输入流和输出流,输入流用来读取这个序列,而输出流则用来写这个序列,在默认情况下 Java 的流操作是基于字节的,即一次只读或只写一个字节。

Java I/O 包提供了 `InputStream` 和 `OutputStream` 作为对 I/O 操作的抽象,这两个接口决定了类层次结构的基本格局。

导致 I/O 性能低下的主要原因是没有对 I/O 操作进行缓冲。众所周知,硬盘擅长于大块数据的读写,但是在小量数据的读写上性能不好,所以,为了最大化 I/O 性能,我们应该选择批量的数据操作,而缓冲流正是为这个目的设计的。缓冲流,包括 `BufferedInputStream` 和 `BufferedOutputStream`,它为 I/O 流增加了内存缓冲区,使得 Java 程序一次可以向底层设备写入或者读取大量数据,从而提高了程序的性能。具体的实现过程中,缓冲流将每一个小的读写请求积攒起来,然后一次性地批处理,通常是将几千个读写请求合并成一个大的请求。

缓冲流虽然在它的内部增加了内存缓冲区,使得在缓冲区和底层设备之间写入或读取大量数据成为可能,但是在这之上的 Java 程序依然使用 `while` 循环从该流(实际上是缓冲数组)按字节读写数据。由于缓冲数组的大小是固定的,JVM 需要针对数组做越界检查,该操作会导致额外的系统开销。另一方面,为了支持多线程环境,将数据从 `BufferedInputStream` 的缓冲数组拷贝到 `BufferedOutputStream` 的缓冲数组,其中的方法调用很多都是 `synchronized`,这也会导致额外的系统开销。我们可以通过利用硬盘擅长读写大块数据的特性,建立自己的缓冲区,避免上述额外的系统开销。注意,使用这个策略需要权衡两个因素。首先是缓冲区的大小,它所创建的缓冲区的大小等于被拷贝的文件大小,当文件很大时,该缓冲区也会很大。第二,它为每次文件拷贝操作都要创建一个新的缓冲区,当有大量文件需要拷贝时,JVM 不得不分配和回收这些大缓冲区内存,

这对程序性能是极大的伤害。

如何在保持速度的前提下避免这两个缺陷呢？方法是这样的：我们可以创建一个静态的固定长度的字节数组，如 1024×1024 字节即 1MB，每次只读写 1MB 的数据。虽然对于大于 1MB 的文件会需要多次读写才能完成整个文件的拷贝，但是这样避免了内存的反复分配和回收。缓冲区大小是可以调整的，针对某个具体的应用场景，我们可以在速度和内存之间取得一个最佳的平衡。

一般情况下，我们总是可以为具体的应用程序找到改善 I/O 性能的方法，这需要具体分析该应用程序的目的和操作特性。例如，考虑 FTP 和 HTTP 服务器，这些服务器的主要工作就是将文件从磁盘拷贝到网络的 Socket，一个网站的主页通常比其他网页访问得更多。为了提高性能，我们可以建立快速缓冲存储区，将那么经常被访问的文件缓存起来，这样这些文件就不必每次从磁盘读写，而是直接从内存拷贝到网络。

3.5.11 Finally 方法里面释放或者关闭资源占用

程序中使用到的资源应当被释放，以避免资源泄漏，这个步骤最好是放在 finally 块中去做。不管程序执行的结果如何，finally 块总是会被执行的，以确保资源的正确关闭。如代码清单 3-141 所示。

代码清单 3-141 释放资源示例

```
import java.io.*;

public class cs {

    public static void main (string args[]) {
        cs cs = new cs ();
        cs.method ();
    }

    public void method () {
        fileinputstream fis = null;
        try {
            fis = new fileinputstream ("cs.java");
            int count = 0;
            while (fis.read () != -1)
                count++;
            system.out.println (count);
        } catch (filenotfoundexception e1) {
        } catch (ioexception e2) {
        } finally{
            fis.close ();
        }
    }
}
```

3.5.12 资源管理机制

对于资源管理，大多数开发人员都知道的一条原则是：谁申请，谁释放。这些资源涉及操作系统中的主存、磁盘文件、网络连接和数据库连接等。凡是数量有限的、需要申请和释放的实体，

都应该纳入到资源管理的范围中来。

对于 C++ 程序员来说，程序的内存管理是他们的一项职责。他们需要保证每一块申请的内存都在正确的地方得到了释放，要么在构造函数中申请，在析构函数中释放，要么使用类似智能指针一样的结构来实现资源管理。Java 语言把内存管理的任务交给了 Java 虚拟机，通过自动垃圾回收机制减少了开发人员的很多工作。但是像输入输出流和数据库连接这样的资源，还是需要开发人员手动释放的。

在使用资源的时候，有可能会抛出各种异常，比如读取磁盘文件和访问数据库时都可能出现各种不同的异常。而资源管理的一个要求就是不管操作是否成功，所申请资源都要被正确释放。通过 try-catch-finally 语句块的 finally 语句进行资源释放操作，这种方式虽然比较易懂，但是其中包含的冗余代码比较多。为了简化这种典型的应用，Java7 对 try 语句进行了增强，使它可以支持对资源进行的管理，保证资源总是被正确释放。即资源的申请是在 try 子句中进行的，而资源的释放则是自动完成的。在使用 try-with-resource 语句的时候，异常可能发生在 try 语句中，也可能发生在释放资源时。如果资源初始化时或 try 语句中出现异常，而释放资源的操作正常执行，try 语句中的异常会被抛出；如果 try 语句和释放资源都出现了异常，那么最终抛出的异常是 try 语句中出现的异常，在释放资源时出现的异常会作为被抑制的异常添加进去，即通过 Throwable.addSuppressed 方法来实现。

能够被 try 语句所管理的资源需要满足一个条件，那就是其 Java 类要实现 java.lang.AutoCloseable 接口，否则会出现编译错误。当需要释放资源的时候，该接口的 close 方法会被自动调用。Java 类库中已有不少接口或类继承或实现了这个接口，使得它们可以用在 try 语句中。在这些已有的常见接口或类中，最常用的就是与 I/O 操作和数据库相关的接口。与 I/O 相关的 java.io.Closeable 继承了 AutoCloseable，而与数据库相关的 java.sql.Connection、java.sql.ResultSet 和 java.sql.Statement 也继承了该接口。如果希望自己开发的类也能利用 try 语句的自动化资源管理，只需要实现 AutoCloseable 接口即可，如代码清单 3-142 所示。

代码清单 3-142 实现 AutoCloseable 接口

```
public class CustomResource implements AutoCloseable{

    @Override
    public void close() throws Exception {
        // TODO Auto-generated method stub
        System.out.println("进行资源释放。");
    }

    public void useCustomResource() throws Exception{
        try(CustomResource resource = new CustomResource()){
            System.out.println("使用资源。");
        }
    }
}
```

当对多个资源进行管理的时候，在释放每个资源时都可能会产生异常。所有这些异常都会被

加到资源初始化异常或 try 语句块中抛出的异常的被抑制异常列表中。在 try-with-resource 语句中也可以使用 catch 和 finally 子句。在 catch 子句中可以捕获 try 语句块和释放资源时可能发生的各种异常。

3.5.13 牺牲 CPU 时间

时间换空间通常用于嵌入式设备，或者内存、硬盘空间不足的情况。通过使用牺牲 CPU 的方式，获得原本需要更多内存或者硬盘空间才能完成的工作。

一个非常简单的时间换空间的算法，实现了 a、b 两个变量的值交换。交换两个变量最常用的方法是使用一个中间变量，而引入额外的变量意味着要使用更多的空间。采用代码清单 3-143 所示的方法可以免去中间变量，而达到变量交换的目的，其代价是引入了更多的 CPU 运算。

代码清单 3-143 免去中间变量方法

```
a=a+b;
b=a-b;
a=a-b;
```

另一个较为有用的例子是对无符号整数的支持。在 Java 语言中，不支持无符号整数，这意味着当需要无符号的 byte 时，需要使用 short 代替，这也意味着空间的浪费。下面代码演示了使用位运算模拟无符号 byte。虽然在取值和设值过程中需要更多的 CPU 运算，但是可以大大降低对内存空间的需求。

代码清单 3-144 无符号整数方式

```
public class UnsignedByte {
    public short getValue(byte i) { // 将 byte 转为无符号的数字
        short li = (short) (i & 0xff);
        return li;
    }

    public byte toUnsignedByte(short i) {
        return (byte) (i & 0xff); // 将 short 转为无符号 byte
    }

    public static void main(String[] args) {
        UnsignedByte ins = new UnsignedByte();
        short[] shorts = new short[256]; // 声明一个 short 数组
        for (int i=0; i<shorts.length; i++) { // 数组不能超过无符号 byte 的上限
            shorts[i] = (short) i;
        }
        byte[] bytes = new byte[256]; // 使用 byte 数组替代 short 数组
        for (int i=0; i<bytes.length; i++) {
            bytes[i] = ins.toUnsignedByte(shorts[i]); // short 数组的数据存到 byte 数组中
        }
        for (int i=0; i<bytes.length; i++) {
```

```
        System.out.println(ins.getValue(bytes[i])+" "); //从 byte 数组中取出无符号
    的 byte
    }
}
}
```

代码 3-144 运行输出如清单 3-145 所示。

代码清单 3-145 3-144 运行输出

```
0
1
2
3
4
5
6
7
8
9
10
11
243
244
245
246
247
248
249
250
251
252
253
254
255
```

如果 CPU 的能力较弱,可以采用牺牲空间的方式提高计算能力,实例如代码清单 3-146 所示。

代码清单 3-146 牺牲空间方式

```
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;

public class SpaceSort {
    public static int arrayLen = 1000000;

    public static void main(String[] args){
        int[] a = new int[arrayLen];
        int[] old = new int[arrayLen];
    }
}
```



```

Map<Integer, Object> map = new HashMap<Integer, Object>();
int count = 0;
while(count < a.length){
    //初始化数组
    int value = (int) (Math.random()*arrayLen*10)+1;
    if(map.get(value)==null){
        map.put(value, value);
        a[count] = value;
        count++;
    }
}
System.arraycopy(a, 0, old, 0, a.length); //从a数组拷贝所有数据到old数组
long start = System.currentTimeMillis();
Arrays.sort(a);
System.out.println("Arrays.sort spend:"+(System.currentTimeMillis() - start)+"ms");
System.arraycopy(old, 0, a, 0, old.length); //恢复 原有数据
start = System.currentTimeMillis();
spaceTotime(a);
System.out.println("spaceTotime spend:"+(System.currentTimeMillis() - start)+
"ms");
}

public static void spaceTotime(int[] array){
    int i = 0;
    int max = array[0];
    int l = array.length;
    for(i=1; i<l; i++){
        if(array[i]>max){
            max = array[i];
        }
    }
    int[] temp = new int[max+1];
    for(i=0; i<l; i++){
        temp[array[i]] = array[i];
    }
    int j = 0;
    int maxl = max + 1;
    for(i=0; i<maxl; i++){
        if(temp[i] > 0){
            array[j++] = temp[i];
        }
    }
}
}

```

函数 `spaceToTime()` 实现了数组的排序。它不计空间成本，以数组的索引下标来表示数据大小，因此避免了数字间的相互比较。这是一种典型的以空间换时间的思路。

3.5.14 对象操作

在 `java.util` 包中新增了一个用来操作对象的工具类 `java.util.Objects`。`Objects` 类中包含的都是静态方法，通过这些方法可以快速对对象进行操作。

在进行两个对象的比较操作时，可以使用 `Objects` 类的 `compare` 方法。一般来说，进行对象比较是先由 Java 类实现 `java.lang.Comparable` 接口，再通过 `compareTo` 方法来进行比较。如果对集合中的元素进行排序，那么还会用到 `java.util.Comparator` 接口的实现。`Objects` 类中的 `compare` 方法可以将两个对象通过特定的 `Comparator` 接口的实现对象来进行比较。

判断对象相等的方式一般是调用 `Object` 类的 `equals` 方法，如判断两个对象 `a` 和 `b` 是否相等，可以使用代码“`a.equals(b)`”。`Objects` 类的 `equals` 方法可以直接判断两个对象是否相等，如“`Objects.equals(a,b)`”。此方法的一个好处是会对 `null` 值进行处理。如果直接调用一个对象的 `equals` 方法，需要先判断这个对象是否为 `null`，而使用 `Objects` 类的 `equals` 方法则不需要。如果 `Objects` 类的 `equals` 方法调用时的两个参数的值都是 `null`，则判断结果是 `true`。第二如果只有一个参数为 `null`，则判断结果是 `false`；如果两个参数都不为 `null`，则调用第一个参数的 `equals` 方法来进行判断。`Objects` 类与 `equals` 方法作用相似的是 `deepEquals` 方法，利用该方法也可以对两个对象进行相等性判断。与 `equals` 方法不同的是，如果 `deepEquals` 方法的两个参数都是数组，则会调用 `java.util.Arrays` 类的 `deepEquals` 来进行比较。`Arrays` 类的 `deepEquals` 方法在进行数组比较时，会考虑数组中的所有元素的相等性。在其他情况下，`deepEquals` 方法和 `equals` 方法的作用是相同的。

`Objects` 类中的 `hashCode` 方法可以用来获取对象的哈希值。如果参数为 `null`，那么返回值是 0；是否返回值是参数对象的 `hashCode` 方法的返回结果。如果需要计算一组对象的哈希值，那么可以使用 `Objects` 类的 `hash` 方法。`Objects` 类的 `hash` 方法的实现使用的是 `Arrays` 类中的 `hashCode` 方法。需要注意的是，在调用 `hash` 方法时传入耽搁对象作为参数的返回结果，与使用同样的参数调用 `hashCode` 方法的结果并不相同。

`Objects` 中还有一组用于获取对象的字符串表示的 `toString` 方法的不同重载形式。`Objects` 的 `toString` 方法在参数为 `null` 时返回值是“`Null`”，而在其他情况下相当于调用参数对象的 `toString` 方法。如果希望在参数为 `null` 时返回给定的内容作为提示信息，那么可以使用 `toString` 方法的另外一个重载形式，即通过一个额外的参数来制定参数值为 `null` 时的返回结果。

3.5.15 正则表达式

Java7 对 `java.util.regex` 包中的内容进行了更新，主要包括以下几个方面。

1. 支持命名捕获分组

捕获分组 (capturing group) 在使用正则表达式从文本中获取某种模式的部分字符时非常有用。通过把感兴趣的字符串的模式封装在捕获分组中，可以在匹配之后很容易地获取这些内容。另外捕获分组也可以在正则表达式中以后向引用 (back reference) 的方式来直接使用，以表示相同的模式。在 Java7 之前，对捕获分组的引用只支持使用表示出现顺序的数字形式。这体现在 `java.util.Matcher` 类的 `group` 方法只接受 `int` 类型作为参数，后向引用的语法也仅支持类似“`\1`”这样的形式。如果一个正则表达式中包含很多捕获分组，那么开发人员需要清除每个数字所代表的捕获分组的含义，这对于代码的编写者和阅读者来说都是一件很麻烦的事情。Java7 引入的命名捕获分组可以很好地解决这个问题。通过为每个捕获分组添加一个有意义的名字，使开发人员可以

很容易地明白每个分组所表示的含义，这比使用无意义的数字要方便很多。

下面代码给出了通过命名捕获分组来匹配字符串并提取内容时的用法。待匹配的字符串是一个 URL，其中通过路径的不同部分来表示查询参数的名称和值。这种采用路径而不是查询字符串来指明参数的方式，在目前的 Web 开发中比较常见。在正则表达式的模式中，为提取每个参数内容的捕获分组都指定了一个有意义的名字。当匹配完成之后，可以通过 `Matcher` 类的 `group` 方法来获取每个捕获分组的内容，参数是在模式中指定的名字。采用“? <”格式为一个捕获分组命名，“<”中的内容是名称。名称必须由大小写英文字母和数字组成，同时第一个字符必须是字母，获取分组的方式如代码清单 3-147 所示。

代码清单 3-147 获取分组方式

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class namedCapturingGroup {
    public static void main(String[] args){
        String url = "http://www.example.org/uid/alex/docid/1/title/MyFirstBlog";
        Pattern pattern = Pattern.compile("^.*uid/(?<uid>.*)/docid/(?<docid>.*)/title/(?<title>.*)");
        Matcher matcher = pattern.matcher(url);
        if(matcher.matches()){
            String uid = matcher.group("uid");
            String docId = matcher.group("docId");
            String title = matcher.group("title");
            System.out.println(uid);
            System.out.println(docId);
            System.out.println(title);
        }
    }
}
```

捕获分组的名称也可以用在正则表达式之中，用来替换使用数字来进行后向引用的做法。

2. 使用“\X”表示 Unicode 中的代码点

在正则表达式中，要引用 Unicode 字符时，可以通过“\uXXX”的形式，如“\u0030”表示数字“0”。通过这种方式可以表示某些无法在源代码中直接出现的字符，比如无法输入汉字的开发人员可以通过这种方式来表示中文字符。如果一个 Unicode 字符不在基本多语言平台（BMP）中，那么要在 Java 中以代理项对的方式出现，在转移成正则表达式中使用“\u”形式的时候，则需要两个相邻的字符。这样既不够直观，使用起来也不方便。Java7 对正则表达式新增了“\x”来直接表示 Unicode 中的代码点。“\x”的使用方式与“\u”类似，只不过允许表示的范围更广，如 Unicode 代码点 U+1011F 可以直接表示成“\x1011F”。

3. 新标记 Pattern.UNICODE_CHARACTER_CLASS

在通过 `Pattern` 类的 `compile` 方法对正则表达式进行编译时，可以指定多个标记。这些标记可以控制匹配时的行为。常见的标记包括设置匹配时不区分大小写的 `Pattern.CASE_INSENSITIV`，

以及设置 “.” 匹配包括换行符在内的所有字符的 `Pattern.DOTALL`。Java7 中添加了一个额外的标记 `Pattern.UNICODE_CHARACTER_CLASS` 来设置使用 Unicode 版本的预定义字符类和 POSIX 字符类。以 “`\d`” 这个预定义字符类为例，在默认情况下，只会匹配 Unicode 规范中所有定义的所有属于数字类别的字符，不只是 0 到 9 的数字，也会包括其他语言中的数字字符。下面代码中，使用 “`(\d+)`” 模式来匹配字符串中的数字。示例中输入的字符串是 “100 1 0 0”，其中包含了一半的数字 “100” 和全角数字 “1 0 0”。在两个 `Pattern` 类的对象中，第一个 `Pattern` 类的对象在编译时没有使用 `UNICODE_CHARACTER_CLASS` 标记，只能匹配一般的数字 “100”，而第二个 `Pattern` 类对象可以匹配整个字符串。

代码清单 3-148 Compile 方式

```
public void useUnicodeCharacterClass() {
    String str = "100 1 0 0";
    Pattern pattern = Pattern.compile("(\\d+)");
    Matcher matcher = pattern.matcher(str);
    if (matcher.find()) {
        String digit = matcher.group(1); // 值为 100
        System.out.println(digit);
    }
    pattern = Pattern.compile("(\\d+)", Pattern.UNICODE_CHARACTER_CLASS);
    matcher = pattern.matcher(str);
    if (matcher.find()) {
        String digit = matcher.group(1); // 值为 100 1 0 0
        System.out.println(digit);
    }
}
```

受到 `UNICODE_CHARACTER_CLASS` 标记影响的字符类除了 “`\d`” 之外，还包括 “`\s`”、“`\w`”、“`\p{Lower}`”、“`\p{Upper}`” 和 “`\p{Punct}`” 等。

4. 指定 Unicode 字符使用的书写格式

Java7 中另外一个与正则表达式相关的更新也与 Unicode 相关。在 Java7 之前，正则表达式在匹配 Unicode 字符串时允许指定 Unicode 字符所在的区块和类别，而在 Java7 中还允许指定 Unicode 字符使用的书写格式 (script)。在指定书写格式时使用的是 “`\p`”，如 “`\p{script=Han}`” 的含义是匹配字符串中书写格式为汉字的 Unicode 字符。在匹配时可以使用的合法书写格式名称都定义在枚举类型 `Character.UnicodeScript` 中。

3.5.16 压缩文件处理

Java 标准库中的 `java.util.zip` 包用于对压缩文件进行处理。Java7 对这一部分功能的更新也比较多。在对文件进行压缩时，允许选择压缩时缓存的中间结果的输出方式，这体现在 `java.util.zip.Deflater` 类的 `deflate` 方法增加了一个参数来表示不同的输出方式。默认的方式是 `Deflater.NO_FLUSH`，该方式由压缩者来确定输出缓存的中间结果的具体时机。在这种方式下，压缩者可以自由决定缓存中数据的大小，因此通常可以获得最佳的性能。第二种输出方式是 `Deflater.SYNC_FLUSH`，这种方式在每次调用 `deflate` 方法时自动清空内部缓冲区，把压缩的中间结果输出。这种方式的好处在于，如果有解压缩程序正在等待压缩之后的输出结果，及时地清空

缓冲区可以让解压缩程序更早地开始工作，有利于提高压缩程序的工作效率。不过这种方式会对压缩性能产生影响。最后一种输出方式是 `Deflater.FULL_FLUSH`，这种方式除了清空缓冲区之外，同时还重置压缩者的内部状态。如果解压缩的程序发现压缩的结果不正确，可以使用此方式来调用 `deflate` 方法，要求压缩者进行压缩操作。这种方式对性能的影响更大，只在必要时才使用。

与 `Deflater` 类对应的 `java.util.zip.DeflaterOutputStream` 类的对象在创建时增加了一个参数 `syncFlush`，用来表示对 `Deflater` 类的对象所缓存的中间结果的处理方式。如果 `syncFlush` 的值为 `true`，那么在调用 `DeflaterOutputStream` 类的对象的 `flush` 方式来清空其本身的内部缓冲区之前，会先按照 `SYNC_FLUSH` 的方式清空对应的 `Deflater` 类的对象所缓存的中间结果。

在 Java7 之前，压缩文件中的文件名和注释都是用默认编码格式 UTF-8。这种 UTF-8 格式可能造成通过 Java 压缩的文件无法被其他工具打开。为了解决这个问题，Java7 允许在创建压缩文件时显式地指定文件名和注释所用的字符集。这体现在 `java.util.zip.ZipFile`、`java.util.zip.ZipInputStream` 和 `java.util.zip.ZipOutputStream` 类的构造方法中都增加了一个 `java.nio.charset.Charset` 类的对象作为参数。这个 `Charset` 类的对象就表明了压缩文件中文件名和注释所用的编码字符集。通过 Java 产生的压缩文件中也包含了相关的元数据，用来表示压缩时的文件名和注释所使用的字符集。

除了上面这些比较大的改动之外，还有一些相关的小改动，包括：Java7 支持大于 4GB 的压缩文件的处理；`ZipFile` 类实现了 `java.io.Closeable` 接口，从而可以在 `try-with-resources` 语句中使用；`ZipFile` 类多了一个方法 `getComment`，用来获取在创建压缩文件时通过 `ZipOutputStream` 类的 `setComment` 方法所添加的文件注释；如果 `java.util.zip.GZIPInputStream` 类在处理压缩文件时遇到了格式不正确的压缩文件，会抛出更加具体的 `java.util.zip.ZipException` 异常。

在集合类方面，`java.util.Collections` 类增加了两个新的方法 `emptyIterator` 和 `emptyEnumeration`，用来返回空的迭代器对象和枚举对象。

3.6 本章小结

本章首先针对面向对象基础、基础类型概念列举了一些优化建议及范例代码，然后对集合类的优化方案，特别是 Java8 的一些新特性进行了解释及范例代码演示。接下来，对字符串操作的优化建议及实践、对象引用级别的优化及实践这两个主题进行深入解释。最后，演示了其他一些方面的优化方案。由于篇幅所限，不能列举所有的优化方案及实践经验，请读者见谅。

4

chapter

第4章 程序设计优化建议

齐使者如梁，孙臆以刑徒阴见，说齐使。齐使以为奇，窃载与之齐。齐将田忌善而客待之。忌数与齐诸公子驰逐重射。孙子见其马足不甚相远，马有上、中、下辈。于是孙子谓田忌曰：“君弟重射，臣能令君胜。”田忌信然之，与王及诸公子逐射千金。及临质，孙子曰：“今以君之下驷与彼上驷，取君上驷与彼中驷，取君中驷与彼下驷。”既驰三辈毕，而田忌一不胜而再胜，卒得王千金。于是忌进孙子于威王。威王问兵法，遂以为师。这正是历史上很有名的“田忌赛马”典故。

任务程序都不可能与设计方案、业务逻辑完全无关，只要你的程序代码需要涉及业务逻辑知识，你就需要采用较好的设计方法，否则性能就会下降。这也正和田忌赛马一样，我们要会取舍，会扬长避短，会吸取他人的优点，这样我们的程序才能在软件设计上达到最优化境界。

本章主要介绍和解决以下问题，程序设计过程涉及整个软件的性能：

- 什么是算法优化建议。
- 如何更好地利用设计模式。
- 如何使用 Java 网络包、如何操作数据库。
- 如何解决海量数据处理、存储问题。
- 如何更好地对程序逻辑进行优化、避免出现问题。
- 关于 Web 系统的优化建议。

4.1 算法优化概述

对应用程序进行性能调优所能获得的最大收益往往来自于算法效率的提高。高效的算法让应用程序使用更少的 CPU 指令、更短的执行路径实现程序功能，也就是说，通常情况下拥有更短执行路径的应用程序运行得更快。缩短执行路径的长度有很多种方法，例如从应用程序的最高层来看，使用更优的数据结构或者改进算法往往可以构造出更短的执行路径。很多应用程序的性能问

题都源于使用了不合适的数据结构，使用恰当的数据结构及算法是提升程序性能最有效的方法。因此，性能分析过程中，要充分注意程序使用的数据结构及算法，尽可能采用更优的方式，才能最大限度地提高程序性能。

4.1.1 常用算法逻辑描述

我们日常编写程序时容易接触到的算法，包括先进先出算法、最近最少使用算法、最近最多使用算法等，JDK 中许多数据结构类的设计、缓存的实现都必须基于这些算法模型，才能让应用软件开发人员可以根据自己的软件业务逻辑选择合适的中间件或者数据结构。

4.1.1.1 FIFO 算法

FIFO（First in First out），即先进先出算法，比如在超市购物之后会提着我们满满的购物车来到收银台排在结账队伍的最后，眼睁睁地看着前面的客户一个个离开，这就是一种先进先出机制，先排队的客户先行结账离开。其实在操作系统的设计理念中很多地方都利用到了先进先出的思想，比如作业调度机制，采用先来先服务的原则，为什么这个原则在很多地方都会用到呢？因为这个原则简单，符合人们的惯性思维，具备公平性，并且实现起来简单，通过直接使用数据结构中的队列即可实现。

FIFO 是队列机制中最简单的，每个接口上都存在 FIFO 队列，表面上看 FIFO 队列并没有提供什么 QoS（Quality of Service，服务质量）保证，甚至很多人认为 FIFO 严格意义上不算做一种队列技术，实则不然，FIFO 是其他队列的基础，FIFO 也会影响到衡量 QoS 的关键指标：报文的丢弃、延时、抖动。既然只有一个队列，自然不需要考虑如何对报文进行复杂的流量分类，也不用考虑下一个报文怎么拿、拿多少的问题，而且因为按顺序取报文，FIFO 无须对报文重新排序。简化了这些实现其实也就提高了对报文时延的保证。

在 FIFO Cache 设计中，核心原则就是，如果一个数据最先进入缓存中，则应该最早被淘汰。也就是说，当缓存满的时候，应当把最先进入缓存的数据给淘汰掉。在设计一个基于 FIFO 算法的 Cache（缓存）组件时应该支持以下操作。

- get(key): 如果 Cache 中存在该 key，则返回对应的 value 值，否则，返回-1。
- set(key,value): 如果 Cache 中存在该 key，则重置 value 值；如果不存在该 key，则将该 key 插入到到 Cache 中，若 Cache 已满，则淘汰最早进入 Cache 的数据。

举个例子，假如 Cache 大小为 3，访问数据序列为 set(1,1)、set(2,2)、set(3,3)、set(4,4)、get(2)、set(5,5)则 Cache 中的数据变化如代码清单 4-1 所示。

代码清单 4-1 Cache 中的数据变化

```
(1,1) set(1,1)
(1,1) (2,2) set(2,2)
(1,1) (2,2) (3,3) set(3,3)
(2,2) (3,3) (4,4) set(4,4)
(2,2) (3,3) (4,4) get(2)
(3,3) (4,4) (5,5) set(5,5)
```

那么利用什么数据结构来这样的实现呢？我们可以利用一个双向链表保存数据，当来了新的

数据之后便添加到链表末尾，如果 Cache 存满数据，则把链表头部数据删除，然后把新的数据添加到链表末尾。在访问数据的时候，如果在 Cache 中存在该数据的话，则返回对应的 value 值，否则返回-1。如果想提高访问效率，可以利用 HashMap¹来保存每个 key 在链表中对应的位置。

JDK 自带的 LinkedHashMap 类是基于 FIFO 实现的，默认情况下 LinkedHashMap 就是按照添加顺序保存，我们只需重写下 removeEldestEntry 方法即可轻松实现一个 FIFO 缓存，简化版的实现代码如代码清单 4-2 所示。

代码清单 4-2 LinkedHashMap 的 FIFO 实现

```
final int cacheSize = 5;
LinkedHashMap<Integer, String> lru = new LinkedHashMap<Integer, String>(){
    @Override
    protected boolean removeEldestEntry(Map.Entry<Integer, String> eldest) {
        return size() > cacheSize;
    }
};
```

FIFO 关心的就是队列长度问题，队列长度会影响到时延、抖动、丢包率。因为队列长度是有限的，有可能被填满，这就涉及该机制的丢弃原则。常见的一个丢弃原则叫作 Tail Drop 机制。简单地讲就是该队列如果已经满了，那么后续进入的报文被丢弃，而没有什么机制来保证后续的报文可以挤掉已经在队列内的报文。在这种机制中，如果定义了较长的队列长度，那么队列不容易填满，被丢弃的报文也就少了，但是队列长度太长了会出现时延的问题，一般情况下时延的增加会导致抖动也增加。如果定义了较短的队列，时延的问题可以得到解决，但是发生 Tail Drop 的报文就变多了。

4.1.1.2 LFU 算法

LFU (Least Frequently Used)，即最近最多使用算法。它是基于“如果一个数据在最近一段时间内使用次数很少，那么在将来一段时间内被使用的可能性也很小”的思路。LFU 算法需要维护一个队列记录所有数据的访问记录，每个数据都需要维护引用计数。LFU 算法需要记录所有数据的访问记录，内存消耗较高；需要基于引用计数排序，性能消耗较高。

LFU 的每个数据块都有一个引用计数，所有数据块按照引用计数排序，具有相同引用计数的数据块则按照时间排序。具体实现如图 4-1 所示。

如图 4-1 所示的操作包括：

- (1) 新加入数据插入到队列尾部（因为引用计数为 1）；

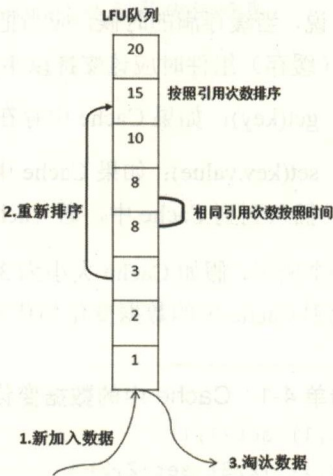


图4-1 LFU算法实现

¹ 基于哈希表的 Map 接口的实现。详见第 3 章里面的介绍。

(2) 队列中的数据被访问后，引用计数增加，队列重新排序；

(3) 当需要淘汰数据时，将已经排序的列表最后的数据块删除。

注意 LFU 和下一小节要介绍的 LRU 算法之间存在的不同之处，LRU 的淘汰规则是基于访问时间，而 LFU 是基于访问次数的。举个简单的例子，假设缓存大小为 3，数据访问序列为 set(2,2)、set(1,1)、get(2)、get(1)、get(2)、set(3,3)、set(4,4)，则在 set(4,4)时对于 LFU 算法应该淘汰(3,3)，而 LRU 应该淘汰(1,1)。LRU 关键是看页面最后一次被使用到发生调度的时间长短，而 LFU 关键是看一定时间段内页面被使用的频率。

那么基于 LFU 算法的 Cache 设计应该支持的操作如。

- get(key): 如果 Cache 中存在该 key，则返回对应的 value 值，否则，返回-1；
- set(key,value): 如果 Cache 中存在该 key，则重置 value 值；如果不存在该 key，则将该 key 插入到到 Cache 中，若 Cache 已满，则淘汰最少访问的数据。

为了能够淘汰最少使用的数据，LFU 算法最简单的一种设计思路就是利用一个数组存储数据项，用 HashMap 存储每个数据项在数组中对应的位置，然后为每个数据项设计一个访问频次，当数据项被命中时，访问频次自增，在淘汰的时候淘汰访问频次最少的数据。这样一来的话，在插入数据和访问数据的时候都能达到 $O(1)$ 的时间复杂度，在淘汰数据的时候，通过选择算法得到应该淘汰的数据项在数组中的索引，并将该索引位置的内容替换为新来的数据内容即可，这样的话，淘汰数据的操作时间复杂度为 $O(n)$ 。

另外还有一种实现思路就是利用最小堆和 HashMap 两者的优势，最小堆中根结点的键值是所有堆结点键值中的最小者。最小堆插入、删除操作都能达到 $O(\log n)$ 时间复杂度，因此效率相比第一种实现方法更加高效。代码清单 4-3 所示的代码是最小堆实现的一个示例。

代码清单 4-3 最小堆实现示例

```
public class SmallHeapDemo {
    final static int MAX_LEN = 100;
    private int queue[] = new int[MAX_LEN];
    private int size;

    public void add(int e){
        if(size >= MAX_LEN)
        {
            System.err.println("over flow");
            return;
        }
        int s = size++;
        shiftup(s,e);
    }

    public int size(){
        return size;
    }
}
```

```

private void shiftup(int s, int e) {
    while(s > 0) {
        int parent = (s - 1) / 2;
        if(queue[parent] < e) {
            break;
        }
        queue[s] = queue[parent];
        s = parent;
    }
    queue[s] = e;
}

public int poll() {
    if(size <= 0)
        return -1;
    int ret = queue[0];
    int s = --size;
    shiftup(0, queue[s]);
    queue[s] = 0;
    return ret;
}

private void shiftdown(int i, int e) {
    int half = size / 2;
    while(i < half) {
        int child = 2 * i + 1;
        int right = child + 1;
        if(right < size && queue[child] > queue[right]) {
            child = right;
        }
        if(e < queue[child]) {
            break;
        }
        queue[i] = queue[child];
        i = child;
    }
    queue[i] = e;
}

public static void main(String args[]) {
    SmallHeapDemo hs = new SmallHeapDemo();
    hs.add(4);
    hs.add(3);
    hs.add(7);
    hs.add(2);
    int size = hs.size();
    for(int i=0; i< size; i++){

```



```

        System.out.println(hs.poll());
    }
}

```

程序运行输出为“2347”。

一般情况下，LFU 效率要优于 LRU，且能够避免周期性或者偶发性的操作导致缓存命中率下降的问题。但 LFU 需要记录数据的历史访问记录，一旦数据访问模式改变，LFU 需要更长时间来适用新的访问模式，即 LFU 存在历史数据影响将来数据的“缓存污染”效用。

4.1.1.3 LRU 算法

LRU 是 Least Recently Used 的缩写，即“最近最少使用”，基于 LRU 算法实现的 Cache 机制简单地说就是缓存一定量的数据，当超过设定的阈值时就把一些过期的数据删除掉，比如我们缓存 10000 条数据，当数据小于 10000 时可以随意添加，当超过 10000 时就需要把新的数据添加进来，同时要把过期数据删除，以确保我们最大缓存 10000 条，那怎么确定删除哪条过期数据呢，采用 LRU 算法实现就是将最老的数据删除。Java 里面实现 LRU 缓存通常有两种选择，一种是使用 LinkedHashMap，一种是自己设计数据结构，使用链表+HashMap 方式。

LinkedHashMap 自身已经实现了顺序存储，默认情况下是按照元素的添加顺序存储，也可以启用按照访问顺序存储，即最近读取的数据放在最前面，最早读取的数据放在最后面，然后它还有一个判断是否删除最老数据的方法，默认是返回 false，即不删除数据。

代码清单 4-4 所示示例是 LinkedHashMap 的一个构造函数，当参数 accessOrder 为 true 时，将会按照访问顺序排序，最后访问的放在最前，最早访问的放在后面。

代码清单 4-4 LRU Cache 的 LinkedHashMap 实现

```

public LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder) {
    super(initialCapacity, loadFactor);
    this.accessOrder = accessOrder;
}

```

代码清单 4-5 所示赛马是 LinkedHashMap 自带的判断方法，判断是否删除最老的元素方法，默认返回 false，即不删除老数据，我们要做的就是重写这个方法，当满足一定条件时删除老数据。

代码清单 4-5 重写删除方法

```

protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
    return false;
}

```

采用 inheritance 方式实现比较简单，该方式实现了 Map 接口，在多线程环境使用时可以使用 Collections.synchronizedMap() 方法实现线程安全操作。

代码清单 4-6 LRU 缓存 LinkedHashMap(inheritance)实现

```

import java.util.LinkedHashMap;
import java.util.Map;

```

```

public class LRUCache2<K, V> extends LinkedHashMap<K, V> {
    private final int MAX_CACHE_SIZE;

    public LRUCache2(int cacheSize) {
        super((int) Math.ceil(cacheSize / 0.75) + 1, 0.75f, true);
        MAX_CACHE_SIZE = cacheSize;
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry eldest) {
        return size() > MAX_CACHE_SIZE;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (Map.Entry<K, V> entry : entrySet()) {
            sb.append(String.format("%s:%s ", entry.getKey(), entry.getValue()));
        }
        return sb.toString();
    }
}

```

代码清单 4-6 的实现是比较标准的实现，在实际使用过程中这样写还是有些烦琐，更实用的方法是像代码清单 4-7 这样写，省去了单独新建一个类的麻烦。

代码清单 4-7 LRU 缓存 LinkedHashMap(inheritance)实现改进版

```

final int cacheSize = 100;
Map<String, String> map = new LinkedHashMap<String, String>((int) Math.ceil
(cacheSize / 0.75f) + 1, 0.75f, true) {
    @Override
    protected boolean removeEldestEntry(Map.Entry<String, String> eldest) {
        return size() > cacheSize;
    }
};

```

相比 inheritance 实现方式来说，delegation 实现方式实现更加优雅一些，但是由于没有实现 Map 接口，所以线程同步就需要自己搞定了。

代码清单 4-8 LRU 缓存 LinkedHashMap(delegation)实现

```

import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Set;

/**
 * Created by liuzhao on 14-5-13.
 */

```



```

public class LRUCache3<K, V> {

    private final int MAX_CACHE_SIZE;
    private final float DEFAULT_LOAD_FACTOR = 0.75f;
    LinkedHashMap<K, V> map;

    public LRUCache3(int cacheSize) {
        MAX_CACHE_SIZE = cacheSize;
        //根据 cacheSize 和加载因子计算 hashmap 的 capacity, +1 确保当达到 cacheSize 上限时
        不会触发 hashmap 的扩容
        int capacity = (int) Math.ceil(MAX_CACHE_SIZE / DEFAULT_LOAD_FACTOR) + 1;
        map = new LinkedHashMap(capacity, DEFAULT_LOAD_FACTOR, true) {
            @Override
            protected boolean removeEldestEntry(Map.Entry eldest) {
                return size() > MAX_CACHE_SIZE;
            }
        };
    }

    public synchronized void put(K key, V value) {
        map.put(key, value);
    }

    public synchronized V get(K key) {
        return map.get(key);
    }

    public synchronized void remove(K key) {
        map.remove(key);
    }

    public synchronized Set<Map.Entry<K, V>> getAll() {
        return map.entrySet();
    }

    public synchronized int size() {
        return map.size();
    }

    public synchronized void clear() {
        map.clear();
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (Map.Entry entry : map.entrySet()) {
            sb.append(String.format("%s:%s ", entry.getKey(), entry.getValue()));
        }
    }
}

```

```

    }
    return sb.toString();
}
}

```

注意，上面的实现方式是非线程安全的，若在多线程环境下使用需要在相关方法上添加 `synchronized` 以实现线程安全操作。

前面说过，除了 `LinkedHashMap` 方式以外，我们还有一种采用 `LRU Cache` 的链表+`HashMap` 实现的方式，如代码清单 4-9 包含的代码所示。

代码清单 4-9 `LRU Cache` 的链表+`HashMap` 实现

```

import java.util.HashMap;

public class LRUCache<K, V> {

    private final int MAX_CACHE_SIZE;
    private Entry first;
    private Entry last;

    private HashMap<K, Entry<K, V>> hashMap;

    public LRUCache(int cacheSize) {
        MAX_CACHE_SIZE = cacheSize;
        hashMap = new HashMap<K, Entry<K, V>>();
    }

    public void put(K key, V value) {
        Entry entry = getEntry(key);
        if (entry == null) {
            if (hashMap.size() >= MAX_CACHE_SIZE) {
                hashMap.remove(last.key);
                removeLast();
            }
            entry = new Entry();
            entry.key = key;
        }
        entry.value = value;
        moveToFirst(entry);
        hashMap.put(key, entry);
    }

    public V get(K key) {
        Entry<K, V> entry = getEntry(key);
        if (entry == null) return null;
        moveToFirst(entry);
        return entry.value;
    }
}

```



```

public void remove(K key) {
    Entry entry = getEntry(key);
    if (entry != null) {
        if (entry.pre != null) entry.pre.next = entry.next;
        if (entry.next != null) entry.next.pre = entry.pre;
        if (entry == first) first = entry.next;
        if (entry == last) last = entry.pre;
    }
    hashMap.remove(key);
}

```

```

private void moveToFirst(Entry entry) {
    if (entry == first) return;
    if (entry.pre != null) entry.pre.next = entry.next;
    if (entry.next != null) entry.next.pre = entry.pre;
    if (entry == last) last = last.pre;

    if (first == null || last == null) {
        first = last = entry;
        return;
    }

    entry.next = first;
    first.pre = entry;
    first = entry;
    entry.pre = null;
}

```

```

private void removeLast() {
    if (last != null) {
        last = last.pre;
        if (last == null) first = null;
        else last.next = null;
    }
}

```

```

private Entry<K, V> getEntry(K key) {
    return hashMap.get(key);
}

```

@Override

```

public String toString() {
    StringBuilder sb = new StringBuilder();
    Entry entry = first;
    while (entry != null) {
        sb.append(String.format("%s:%s ", entry.key, entry.value));
        entry = entry.next;
    }
}

```

```

    }
    return sb.toString();
}

class Entry<K, V> {
    public Entry pre;
    public Entry next;
    public K key;
    public V value;
}
}

```

4.1.2 多核算法优化原理

4.1.2.1 锁机制设计简述

多核算法优化的目标大致有两种，即 lock-free 和 lock-less。lock-free 是完全无锁的设计，具体有两种实现方式。

- (1) Per-Cpu Data，顾名思义，每个核或者线程都有自己私有的数据结构，这里的私有是逻辑上私有，并不意味着别的线程无法访问这些数据，而 thread local data 是线程私有的数据结构，别的线程是无法访问的。当然，不管是逻辑上私有，还是物理上私有，把共享数据转化成线程私有数据就可以避免使用锁，进而避免竞争。全局变量是共享的，而局部变量是私有的，所以多使用局部变量，同样可以达到无锁的目的。
- (2) CAS based，CAS 的全称是 Compare And Swap，属于一个原子操作。自旋锁（Spinlock）是专为防止多处理器并发而引入的一种锁，具体我们会在第 5 章节介绍。自旋锁的实现同样需要 Compare And Swap 方式，但区别是自旋锁只有两个状态 LOCKED 和 UNLOCKED，而 CAS 的变量可以有多个状态。其次，CAS 的实现必须由硬件来保障其原子操作的实现，CAS 一次可以操作 32bits，即一次可以比较、修改一块内存。基于 CAS 实现的数据结构没有一个统一、一致的实现方法，所以有时不如基于锁的算法那么简单、直接。针对不同的数据结构有不同的 CAS 实现方法，这里不多做介绍，读者可以找分布式方面的书籍进一步了解。

lock-less 的目的是减少锁的争用（contention），而不是减少锁的使用。这个和锁的粒度（granularity）相关，锁的粒度越小，等待的时间就越短，并发的时间就越长。锁的争用需要考虑不同线程在获取锁后，会执行哪些不同的动作。以资源池的分配和释放管理为例，假设多个线程都会访问同一个资源池，分配或者释放这些线程需要占有的资源。资源池可以是一个双向链表，分配在头部进行，而释放在尾部进行。如果多个线程同时访问资源池，需要一个自旋锁来保护这个资源池。那么分配和释放两个不同的动作，相互之间就会有争用，而且多个线程上的分配，或者释放本身也存在锁争用。

正如前一段描述的，我们可以考虑分配用一个锁，释放用一个锁，生成一个双端队列，这样可以减少分配和释放之间的争用。也可以考虑采用两个资源池，分配请求占据一个资源池，释放请求占据一个资源池，那么当分配资源池所有的资源用完之后，交换两个资源池的指针，这时要

考虑两个资源池都是空的情况，注意这里只是减少了分配和释放的争用，不能完全消除这类争用情况的发生。

不管是 lock-based 还是 CAS-based (lock-free) 的数据结构，都需要一个状态位来统一标识。即让程序在不同状态下做不同的事，而增大锁的粒度，也就是增加了资源池的数量，减小了状态保护的范围。

4.1.2.2 多线程简述

线程是一个程序内部的顺序控制流，一个进程相当于一个任务，一个线程相当于一个任务中的一条执行路径。Java 程序需要最大化地运用多核能力，如果使用多线程方式，只有运行的线程数比核数大，才有可能榨干 CPU 资源，否则会有若干核闲置。需要注意的是，如果线程数目太多，就会占用过多内存，导致性能不升反降。此外，JVM 的垃圾回收也是需要线程的，所以这里的线程数包含 JVM 自己的线程。每个线程有自己的工作内存，在这个区域内，系统可以毫无顾忌的优化，如果去读共享内存区域，性能也不会下降。但是一旦线程想写共享内存(使用 volatile 关键字)，就会插入很多内存屏障操作 (Memory Barrier 或者 Memory Fence) 指令，保证处理器不乱序执行。相比写本地线程自有的变量，性能下降很多。处理方法是尽量减少共享数据，这样也符合“数据耦合”的设计原则。在 Java1.5 中，Synchronize 是性能低效的。因为这是一个重量级操作，需要调用操作接口，导致有可能加锁消耗的系统时间比加锁以外的操作还多。相比之下使用 Java 提供的 Lock 对象性能更高一些。但是 Java1.6 发生了变化。Synchronize 在语义上很清晰，可以进行很多优化，有适应自旋、锁消除、锁粗化、轻量级锁、偏向锁等。导致在 Java1.6 上 Synchronize 的性能并不比 Lock 差。这些相关知识会在第 5 章详细讲解。

4.1.2.3 NUMA 架构

在 NUMA 架构出现前，CPU 欢快地朝着频率越来越高的方向发展。受到物理极限的挑战，又转为核数越来越多的方向发展。如果每个核心的工作性质都是 share-nothing(类似于 Map-Reduce 的各个节点的作业属性)，那么也许就不会有 NUMA。由于所有 CPU 核都是通过共享一个北桥来读取内存，随着核数如何的发展，北桥在响应时间上的性能瓶颈越来越明显。于是，聪明的硬件设计师们，先到了把内存控制器 (原本北桥中读取内存的部分) 也做个拆分，平分到了每个 die 上。

NUMA 是一个 CPU 的特性，NUMA 中，虽然内存直接附属在 CPU 上，但是由于内存被平均分配在了各个通道上。只有当 CPU 访问自身直接附属内存对应的物理地址时，才会有较短的响应时间 (后称 Local Access)。而如果需要访问其他 CP 附属的内存的数据时，就需要通过 inter-connect 通道访问，响应时间就相比之前变慢了 (后称 Remote Access)。所以 NUMA (Non-Uniform Memory Access) 就此得名。

NUMA 中，虽然内存直接附属在 CPU 上，但是由于内存被平均分配在了各个通道上。只有当 CPU 访问自身直接附属内存对应的物理地址时，才会有较短的响应时间 (后称 Local Access)。而如果需要访问其他 CPU attach 的内存的数据时，就需要通过 inter-connect 通道访问，响应时间就相比之前变慢了 (后称 Remote Access)。所以 NUMA (Non-Uniform Memory Access) 就此得名。

从系统架构来说，目前的主流企业服务器基本可以分为三类：SMP (Symmetric Multi Processing, 对称多处理架构)、NUMA (Non-Uniform Memory Access, 非一致存储访问架构)，和 MPP (Massive

Parallel Processing, 海量并行处理架构)。

SMP 架构下 CPU 的核是对称,但是他们共享一条系统总线。所以 CPU 多了之后总线就会成为瓶颈。在 NUMA 架构下,若干 CPU 组成一个组,组之间有点对点的通讯,相互独立。对于用 C/C++ 等开发的程序来说,因为程序直接决定了内存的访问模式,对编程者而言,就需要对 NUMA 架构有所了解,以最大的利用 NUMA 带来的优势,避免反被它伤害。但对 Java 应用来说,因为代码不会直接执行,一定是通过 JVM 进行,所以很大程度上 Java 应用的性能就取决于 JVM 了。

MPP 则是逻辑上将整个系统划分为多个节点,每个节点的处理器只可以访问本身的本地资源,是完全无共享的架构。节点之间的数据交换需要软件实施。它的优点是可扩展性非常好;缺点是彼此数据交换困难,需要控制软件的大量工作来实现通讯以及任务的分配、调度,对于一般的企业应用而言过于复杂,效率不高。

NUMA 架构则在某种意义上是综合了 SMP 和 MPP 的特点:逻辑上整个系统也是分为多个节点,每个节点可以访问本地内存资源,也可以访问远程内存资源,但访问本地内存资源远远快于远程内存资源。它的优点是兼顾了 SMP 和 MPP 的特点,易于管理,可扩充性好;缺点是访问远程内存资源的所需时间非常的大。在实际系统中使用比较广的是 SMP 和 NUMA 架构。像传统的英特尔 IA 架构就是 SMP,而很多大型机采用了 NUMA 架构。

现在已经进入了多核时代,随着核数的越来越多,对于内存吞吐量和延迟有了更高的要求。正是考虑到这种需求,NUMA 架构出现在了最新的英特尔下一代 Xeon 处理器中。目前,Windows Server 2003 和 Windows XP 64-bit Edition, Windows XP 等都是 NUMA aware 的,而 Windows Vista 则有了对 Numa 调度的支持。所有使用 2.6 版本以上 kernel 的 Linux 操作系统都能够支持 NUMA。而 Solaris、HP-Unix 等 UNIX 操作系统也是充分支持 NUMA 架构的。对于数据库产品来说,Oracle 从 8i 开始支持 NUMA,而之后的 Oracle9i、Oracle10g、Oracle11g 都能够支持 NUMA。SQL Server 2005 和 SQL Server 2008 均有效地提供了对 NUMA 的支持。

总的来说,其实无论是 NUMA 还是 Linux Kernel,或者是程序开发他们都没有错,只是还做得不够极致。如果 NUMA 在硬件级别可以提供更多低成本的 profile 接口;如果 Linux Kernel 可以使用更科学的动态调整策略;如果程序开发人员更懂 NUMA,那么我们完全可以更好地发挥 NUMA 的性能,使得无限横向扩展 CPU 核数不再是一个梦想。

4.1.3 Java 算法优化实践

4.1.3.1 冒泡算法优化

冒泡排序 (Bubble Sort),是一种计算机科学领域的较简单的排序算法。它重复地走访过要排序的数列,一次比较两个元素,如果它们的顺序错误就把它交换过来。走访数列的工作是重复地进行直到没有再需要交换,也就是说该数列已经排序完成。针对冒泡排序存在的时间复杂度较高的缺点,设计了一个优化方案。在该优化方案中,我们引入一个标志位 (默认为 true),如果本次或者本趟遍历前后数据比较发生了交换,则标志位设置为 true,否则为 false,直到又一次数据没有发生交换,说明排序完成。程序如代码清单 4-10 所示。

代码清单 4-10 冒泡算法优化方案

```

import java.util.Random;

public class BubbleSortTuningDemo {
    public static void main(String[] args) {

        //构造数据
        int[] arr = constructDataArray(15);
        System.out.println("-----排序前-----");
        printArrayData(arr);
        //冒泡排序
        bubbleSort4(arr);
        System.out.println("-----排序后-----");
        printArrayData(arr);
    }

    //构造数据
    public static int[] constructDataArray(int length){
        int[] arr = new int[length];
        Random random = new Random();
        for(int i=0;i<length;i++){
            arr[i] = random.nextInt(length);
        }
        return arr;
    }

    /**
     * 引入标志位，默认为 true
     * 如果前后数据进行了交换，则为 true，否则为 false。如果没有数据交换，则排序完成。
     * @param arr
     */
    public static int[] bubbleSort4(int[] arr){
        boolean flag = true;
        int n = arr.length;
        while(flag){
            flag = false;
            for(int j=0;j<n-1;j++){
                if(arr[j] > arr[j+1]){
                    //数据交换
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                    //设置标志位
                    flag = true;
                }
            }
            n--;
        }
    }
}

```

```

        return arr;
    }

    //打印数据
    public static void printArrayData(int[] arr){
        for(int d :arr){
            System.out.print(d + " ");
        }
        System.out.println();
    }
}

```

代码清单 4-10 的运行输出如清单 4-11 所示。

代码清单 4-11 运行输出

```

-----排序前-----
0  7  7  2  8  10  3  3  12  6  10  8  2  13  11
-----排序后-----
0  2  2  3  3  6  7  7  8  8  10  10  11  12  13

```

4.1.3.2 Arrays 类中的排序算法

Arrays 类中主要有二分法查找 (binarySearch 方法) 算法和归并排序 (sort 方法) 两种。

二分法查找算法的算法思想决定了当数据量很大适宜采用该方法。采用二分法查找时，数据需是排好序的。基本思想是假设数据是按升序排序的，对于给定值 x，从序列的中间位置开始比较，如果当前位置值等于 x，则查找成功；若 x 小于当前位置值，则在数列的前半段中查找；若 x 大于当前位置值则在数列的后半段中继续查找，直到找到为止。源代码如代码清单 4-12 所示。

代码清单 4-12 二分法查找算法

```

//针对 int 类型数组的二分法查找，key 为要查找数的下标
private static int binarySearch0(int[] a, int fromIndex, int toIndex, int key) {
    int low = fromIndex;
    int high = toIndex - 1;
    while (low <= high){
        int mid = (low + high) >>> 1; //无符号左移一位，相当于除以二
        int midVal = a[mid];
        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1); // key not found.
}

```


Arrays 的排序方法是这样的:

- (1) 如果数组元素的个数小于 7, 直接插入排序。我不知道这个 7 是不是实践当中的最佳参数, 其实我认为小于 10 都行。毕竟快排是需要递归实现的, 每次递归都需要一定的开销。
- (2) 数组元素的个数大于 7, 快速排序。快速排序每一步中, 需要选取一个元素作为 pivot 来划分子序列。最理想的状态就是每次选取的 pivot 能将序列均分为两个长度差不多的子序列。jdk 中排序算法是这样选择 pivot 的: 对一个长度为 len 的数组 a, 选择 a[0]、a[len/8]、a[2*len/8]、a[3*len/8]、a[4*len/8]、a[5*len/8]、a[6*len/8]、a[7*len/8]、a[8*len/8] 这 9 个数的中间值作为 pivot。

sort() 方法针对引用类型数组采取的算法是归并排序。算法思想是, 归并 (Merge) 排序法是将两个 (或两个以上) 有序表合并成一个新的有序表, 即把待排序序列分为若干个子序列, 每个子序列是有序的。然后再把有序子序列合并为整体有序序列。

sort() 方法针对整形数据采取的是快速排序算法, 算法思想: 通过一趟排序将要排序的数据分割成独立的两部分, 其中一部分的所有数据都比另外一部分的所有数据都要小, 然后再按此方法对这两部分数据分别进行快速排序, 整个排序过程可以递归进行, 以此达到整个数据变成有序序列。代码如代码清单 4-13 所示。

代码清单 4-13 归并排序算法

```
public class ArraySourceCode {
    /**
     * Swaps x[a] with x[b].
     */
    private static void swap(int x[], int a, int b) {
        int t = x[a];
        x[a] = x[b];
        x[b] = t;
    }

    public static void sort(int[] a) {
        sort1(a, 0, a.length);
    }

    private static int med3(int x[], int a, int b, int c) { // 找出三个中的中间值
        return (x[a] < x[b] ?
            (x[b] < x[c] ? b : x[a] < x[c] ? c : a) :
            (x[b] > x[c] ? b : x[a] > x[c] ? c : a));
    }

    /**
     * Sorts the specified sub-array of integers into ascending order.
     */
    private static void sort1(int x[], int off, int len) {
        // Insertion sort on smallest arrays
```

```

    if (len < 7) { //采用冒泡排序
        for (int i=off; i<len+off; i++)
            for (int j=i; j>off && x[j-1]>x[j]; j--)
                swap(x, j, j-1);
        return;
    }
    //采用快速排序
    // Choose a partition element, v
    int m = off + (len >> 1); // Small arrays, middle element
    if (len > 7) {
        int l = off;
        int n = off + len - 1;
        if (len > 40) { // Big arrays, pseudomedian of 9
            int s = len/8;
            l = med3(x, l, l+s, l+2*s);
            m = med3(x, m-s, m, m+s);
            n = med3(x, n-2*s, n-s, n);
        }
        m = med3(x, l, m, n); // Mid-size, med of 3
    }
    int v = x[m];

    // Establish Invariant: v* (<v)* (>v)* v*
    int a = off, b = a, c = off + len - 1, d = c;
    while(true) {
        while (b <= c && x[b] <= v) {
            if (x[b] == v)
                swap(x, a++, b);
            b++;
        }
        while (c >= b && x[c] >= v) {
            if (x[c] == v)
                swap(x, c, d--);
            c--;
        }
        if (b > c)
            break;
        swap(x, b++, c--);
    }

    // Swap partition elements back to middle
    int s, n = off + len;
    s = Math.min(a-off, b-a );
    vecswap(x, off, b-s, s);
    s = Math.min(d-c, n-d-1);
    vecswap(x, b, n-s, s);

    // Recursively sort non-partition-elements

```



```
if ((s = b-a) > 1)
    sort1(x, off, s);
if ((s = d-c) > 1)
    sort1(x, n-s, s);
}
```

针对 double、float 类型数组排序的 sort()方法，采取了先把所有的数组元素值为-0.0d 的元素转换成 0.0d，再利用快速排序排好序，最后再还原。

此外，有别于 Arrays 类，LinkedList 类的排序不是采用归并实现的，而是把元素复制到数组后，再调用快速排序。

4.1.3.3 DualPivotQuicksort 类

在 JDK7 中新增了 java.util.DualPivotQuicksort 这个类，里面实现于 2009 年发表的 Dual-Pivot Quicksort 算法。其主要的设计是改进了 Quicksort 算法²，使之效率大幅提高，因此 Collections.sort()、Arrays.sort()等的实现部分使用了这个类。

一般的快速排序采用一个枢轴来把一个数组划分成两半，然后递归计算，大量经验数据表面，采用两个枢轴来划分成 3 份的算法更高效，因此 DualPivotQuicksort 类采用的是这种方式。这种算法需要选举出两个枢轴 P1 和 P2，需要 3 个指针 L、K、G，3 个指针的作用如图 4-2 所示。

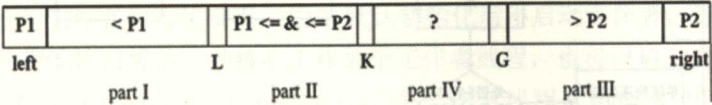


图4-2 DualPivotQuicksort方式

算法步骤按照下面的 7 点顺序依次进行。

- (1) 小于 27 的数组，使用插入排序（或 47）。
- (2) 选择枢轴 P1 和 P2（假设使用数组头和尾）。
- (3) P1 需要小于 P2，否则交换。现在数组被分成 4 份，left 到 L 的小于 P1 的数，L 到 K 的大于 P1 小于 P2 的数，G 到 rigth 的大于 P2 的数，待处理的 K 到 G 的中间的数（逐步被处理到前 3 个区域中）。
- (4) L 从开始初始化直至不小于 P1，K 初始化为 L-1，G 从结尾初始化直至不大于 P2。K 是主移动的指针，来一步一步吞噬中间区域。
 - ****当大于 P1 小于 P2，K++。
 - ****当小于 P1，交换 L 和 K 的数，L++，K++。
 - ****当大于 P2，如果 G 的数小于 P1，把 L 上的数放在 K 上，把 G 的数放在 L 上，L++，再把 K 以前的数放在 G 上，G--，K++，完成一次 L、K、G 的互相交换。否则交换 K

² 快速排序（Quicksort）是对冒泡排序的一种改进。快速排序由 C. A. R. Hoare 在 1962 年提出。

和 G，并 G--，K++。

- (5) 递归 4。
- (6) 交换 P1 到 L-1 上。交换 P2 到 G+1 上。
- (7) 递归。

上述流程的流程图如图 4-3 所示。

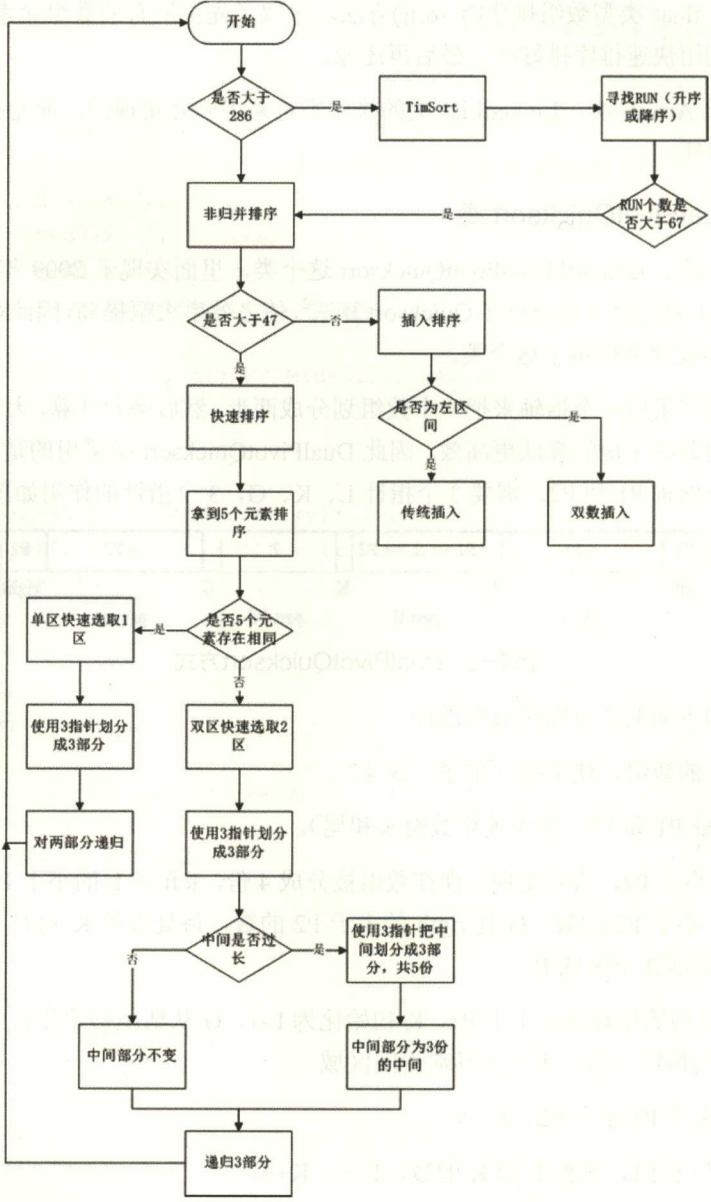


图4-3 DualPivotQuicksort流程图

4.1.3.4 线程池实现算法分析

线程池是一种多线程处理形式，处理过程中将任务添加到队列，然后在创建线程后自动启动这些任务。线程池线程都是后台线程。每个线程都使用默认的堆栈大小，以默认的优先级运行，

并处于多线程单元中。如果某个线程在托管代码中空闲（如正在等待某个事件），则线程池将插入另一个辅助线程来使所有处理器保持繁忙。如果所有线程池线程都始终保持繁忙，但队列中包含挂起的工作，则线程池将在一段时间后创建另一个辅助线程但线程的数目永远不会超过最大值。超过最大值的线程可以排队，但他们要等到其他线程完成后才启动。

我们在编写 Java 程序的过程中也经常使用到线程池技术，它的优点包括如下 3 点。

- **复用**：类似 Web 服务器等系统，长期来看内部需要使用大量的线程处理请求，而单次请求响应时间通常比较短，此时 Java 基于操作系统的本地调用方式大量的创建和销毁线程本身会成为系统的一个性能瓶颈和资源浪费。若使用线程池技术可以实现工作线程的复用，即一个工作线程创建和销毁的生命周期期间内可以执行处理多个任务，从而总体上降低线程创建和销毁的频率和时间，提升了系统性能。
- **流控**：服务器资源有限，超过服务器性能的过高并发设置反而成为系统的负担，造成 CPU 大量耗费于上下文切换、内存溢出等后果。通过线程池技术可以控制系统最大并发数和最大处理任务量，从而很好地实现流控，保证系统不至于崩溃。
- **功能**：JDK 的线程池实现的非常灵活，并提供了很多功能，一些场景基于功能的角度会选择使用线程池。

JDK、Jetty、Tomcat，这几种编程模型或者容器分别都实现了线程池，但是它们所采用的算法是不一样的。

对于线程池构造与工作者初始化步骤，JDK 默认初始化后不启动工作者，等待有请求时才启动。可以通过调用线程池接口提前启动核心工作数个工作者线程，也可以启动业务期望的多个工作者线程。Jetty 6 初始化后直接启动 `_minThreads` 个工作者线程。Jetty 8 初始化后直接启动 `_minThreads` 个工作者线程。Tomcat 基于 JDK 线程池的构造方法。

对于工作者线程存储及并发管理，JDK 使用了 `HashSet` 来存储工作者线程，通过可重入锁 `ReentrantLock` 对其进行并发保护。每个 worker 都是一个 `Runnable` 接口。Jetty 6 同样使用了 `HashSet` 存储工作者线程，通过 `synchronized` 一个对象进行 `HashSet` 的并发保护。每个工作者实际上是一个 `Thread` 的扩展。Jetty 8 使用了 `ConcurrentLinkedQueue` 存储工作者 `workers`，利用 JDK 基于 CAS 算法的实现提高了并发效率，同时也降低了线程池并发保护的复杂程度。针对队列 `ConcurrentLinkedQueue` 无法保证 `size()` 实时性问题引入原子变量 `AtomicInteger` 统计工作者数量。Tomcat 是基于 JDK 的 `ThreadPoolExecutors` 实现，复用 JDK 业务。

对于待处理工作队列结构，JDK 使用了实现接口 `BlockingQueue` 的阻塞队列来存储待处理工作 `job`，并把队列作为构造函数参数，从而实现业务可以灵活的扩展定制线程池的队列。业务也可使用 JDK 自身的同步阻塞队列 `SynchronousQueue`、有界队列 `ArrayBlockingQueue`、无界队列 `LinkedBlockingQueue`、优先级队列 `PriorityBlockingQueue`。Jetty 6 使用了数组存储待处理的 `job` 对象 `Runnable`。数组初始化容量为 `_maxThreads` 个，使用变量 `_queued` 计算保存当前内部待处理 `job` 的个数即数组 `length`。超过数组最大值时，扩大 `_maxThreads` 个容量，因此数组永远够用够大，容量无界。同样是用 `synchronized` 一个对象的方式实现同步。Jetty 8 与 JDK 相同实现，使用了基于接口 `BlockingQueue` 的阻塞队列来存储待处理工作 `job`，也支持在线程池构造函数的参数中传入队列类型。同时，Jetty 8 内部默认未设置队列类型场景可自动设置使用 2 种队列：有界无法扩容的

ArrayBlockingQueue 及 Jetty 自身定制扩展实现的可扩容队列 BlockingArrayQueue。Tomcat 复用了 JDK 方式。

对比几种线程池实现，JDK 的实现是最为灵活、功能最强且扩展性最好的，Tomcat 即基于 JDK 线程池功能扩展实现，复用原有业务的同时扩充了自己的业务。Jetty 6 是完全自己定制的线程池业务，耦合线程池众多复杂的业务逻辑到线程池类里面，逻辑相对最为复杂，扩展性也非常差。Jetty 8 相对 Jetty 6 的实现简化了很多，其中利用了 JDK 中的同步容器和原子变量，同时实现方式也越来越接近 JDK。

4.2 设计模式

设计模式总的来说就是对待特定问题的成熟的解决方案，如果能够合理地使用设计模式，会让整个代码工程显得更加有成效，但是如果肆意乱用，也会起到反作用。笔者在面试过程中会询问设计模式的使用，但是很少有学生接触过，这确实是我们的教育出了问题。教育的目的是为了就业，教授也需要有很强的实践动手能力，而不是理论知识，再高大上的理论，对于大多数学生而言没有多大用处。

4.2.1 设计模式的六大准则

1. 开闭原则（Open Close Principle）

在软件的生命周期内，因为变化、升级和维护等原因需要对软件原有代码进行修改时，可能会给旧代码中引入错误，也可能会使我们不得不对整个功能进行重构，并且需要原有代码经过重新测试。如何解决这类问题呢？解决方法是当软件需要变化时，尽量通过扩展软件实体的行为来实现变化，而不是通过修改已有的代码来实现变化。这就是开闭原则。

开闭原则就是说对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是，“为了使程序的扩展性好，易于维护和升级”。想要达到这样的效果，我们需要使用接口和抽象类。

开闭原则无非就是想表达这样一层意思，用抽象构建框架，用实现扩展细节。因为抽象灵活性好，适应性广，只要抽象的合理，可以基本保持软件架构的稳定。而软件中易变的细节，我们用从抽象派生的实现类来进行扩展，当软件需要发生变化时，我们只需要根据需求重新派生一个实现类来扩展就可以了。当然前提是我们的抽象要合理，要对需求的变更有前瞻性和预见性才行。

2. 里氏代换原则³（Liskov Substitution Principle）

假设一个问题，有一功能 P1，由类 A 完成。现需要将功能 P1 进行扩展，扩展后的功能为 P，其中 P 由原有功能 P1 与新功能 P2 组成。新功能 P 由类 A 的子类 B 来完成，则子类 B 在完成新功能 P2 的同时，有可能会致原有功能 P1 发生故障。解决方法是当使用继承时，遵循里氏替换原则。类 B 继承类 A 时，除添加新的方法完成新增功能 P2 外，尽量不要重写父类 A 的方法，也尽量不要重载父类 A 的方法。

³ 1988 年由麻省理工学院的 Barbara Liskov 女士所提出。

里氏代换原则 (Liskov Substitution Principle LSP) 面向对象设计的基本原则之一。里氏代换原则中说, 任何基类可以出现的地方, 子类一定可以出现。LSP 是继承复用的基石, 只有当衍生类可以替换掉基类, 软件单位的功能不受到影响时, 基类才能真正被复用, 而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现, 所以里氏代换原则是对实现抽象化的具体步骤的规范。

里氏替换原则通俗的来讲就是: 子类可以扩展父类的功能, 但不能改变父类原有的功能。它包含以下4层含义:

- 子类可以实现父类的抽象方法, 但不能覆盖父类的非抽象方法。
- 子类中可以增加自己特有的方法。
- 当子类的方法重载父类的方法时, 方法的前置条件 (即方法的形参) 要比父类方法的输入参数更宽松。
- 当子类的方法实现父类的抽象方法时, 方法的后置条件 (即方法的返回值) 要比父类更严格。

3. 依赖倒转原则 (Dependence Inversion Principle)

这个是开闭原则的基础, 具体内容: 真对接口编程, 依赖于抽象而不依赖于具体。采用依赖倒置原则可以减少类间的耦合性, 提高系统的稳定性, 减少并行开发引起的风险, 提高代码的可读性和可维护性。

依赖倒转原则的本质就是通过抽象 (接口或抽象类) 使各个类或模块的实现彼此独立, 不互相影响, 实现模块间的松耦合, 我们怎么在项目中使用这个规则呢? 需要遵循以下的几个规则。

- 每个类尽量都有接口或抽象类, 或者抽象类和接口两者都具备

这是依赖倒置的基本要求, 接口和抽象类都是属于抽象的, 有了抽象才可能依赖倒置。

- 变量的显示类型尽量是接口或者是抽象类

很多书上说变量的类型一定要是接口或者是抽象类, 这个有点绝对化了, 比如一个工具类, xxxUtils 一般是不需要接口或是抽象类的。还有, 如果你要使用类的 clone 方法, 就必须使用实现类, 这个是 JDK 提供一个规范。

- 任何类都不应该从具体类派生

如果一个项目处于开发状态, 确实不应该有从具体类派生出的子类的情况, 但这也不是绝对的, 因为人都是会犯错误的, 有时设计缺陷是在所难免的, 因此只要不超过两层的继承都是可以忍受的。特别是做项目维护的同志, 基本上可以不考虑这个规则, 为什么? 维护工作基本上都是做扩展开发, 修复行为, 通过一个继承关系, 覆写一个方法就可以修正一个很大的 Bug, 何必再要去继承最高的基类呢?

- 尽量不要覆写基类的方法

如果基类是一个抽象类, 而且这个方法已经实现了, 子类尽量不要覆写。类间依赖的是抽象, 覆写了抽象方法, 对依赖的稳定性会产生一定的影响。

■ 结合里氏替换原则使用

接口负责定义 `public` 属性和方法，并且声明与其他对象的依赖关系，抽象类负责公共构造部分的实现，实现类准确的实现业务逻辑，同时在适当的时候对父类进行细化。

4. 接口隔离原则 (Interface Segregation Principle)

这个原则的意思是说，使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。

很多人会觉得接口隔离原则跟单一职责原则很相似，其实不然。其一，单一职责原则上注重的是职责；而接口隔离原则注重对接口依赖的隔离。其二，单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节；而接口隔离原则主要约束接口，主要针对抽象，针对程序整体框架的构建。

采用接口隔离原则对接口进行约束时，要注意以下几点：

- 接口尽量小，但是要有限度。对接口进行细化可以提高程序设计灵活性是不争的事实，但是如果过小，则会造成接口数量过多，使设计复杂化。所以一定要适度。
- 为依赖接口的类定制服务，只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。只有专注地为一个模块提供定制服务，才能建立最小的依赖关系。
- 提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情。

运用接口隔离原则，一定要适度，接口设计的过大或过小都不好。设计接口的时候，只有多花些时间去思考和筹划，才能准确地实践这一原则。

5. 迪米特法则⁴ (最少知道原则, Demeter Principle)

为什么叫最少知道原则，就是说，一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。比如，一个类公开的 `public` 属性或方法越多，修改时涉及的面也就越大，变更引起的风险扩散也就越大。因此，为了保持朋友类间的距离，在设计时需要反复衡量：是否还可以再减少 `public` 方法和属性，是否可以修改为 `private`、`package-private`（包类型，在类、方法、变量前不加访问权限，则默认为包类型）、`protected` 等访问权限，是否可以加上 `final` 关键字等。

如果一个系统符合迪米特法则，那么当其中某一个模块发生修改时，就会尽量少地影响其他模块，扩展会相对容易，这是对软件实体之间通信的限制，迪米特法则要求限制软件实体之间通信的宽度和深度。迪米特法则可降低系统的耦合度，使类与类之间保持松散的耦合关系。

迪米特法则要求我们在设计系统时，应该尽量减少对象之间的交互，如果两个对象之间不必彼此直接通信，那么这两个对象就不应当发生任何直接的相互作用，如果其中的一个对象需要调用另一个对象的某一个方法的话，可以通过第三者转发这个调用。简言之，就是通过引入一个合理的第三者来降低现有对象之间的耦合度。

⁴ 来自于 1987 年美国东北大学 (Northeastern University) 一个名为 “Demeter” 的研究项目。

在将迪米特法则运用到系统设计中时，要注意下面的几点：

- 在类的划分上，应当尽量创建松耦合的类，类之间的耦合度越低，就越有利于复用，一个处在松耦合中的类一旦被修改，不会对关联的类造成太大波及。
- 在类的结构设计上，每一个类都应当尽量降低其成员变量和成员函数的访问权限；在类的设计上，只要有可能，一个类型应当设计成不变类。
- 在对其他类的引用上，一个对象对其他对象的引用应当降到最低。

注意，迪米特法则的核心观念就是类间解耦，弱耦合，只有弱耦合了以后，类的复用率才可以提高。

6. 合成复用原则（Composite Reuse Principle）

原则是尽量使用合成/聚合的方式，而不是使用继承。只有当以下的条件全部被满足时，才应当使用继承关系。

- (1) 子类是超类的一个特殊种类，而不是超类的一个角色，也就是区分“Has-A”和“Is-A”。只有“Is-A”关系才符合继承关系，“Has-A”关系应当使用聚合来描述。
- (2) 永远不会出现需要将子类换成另外一个类的子类的情况。如果不能肯定将来是否会变成另外一个子类的话，就不要使用继承。
- (3) 子类具有扩展超类的责任，而不是具有置换掉或注销掉超类的责任。如果一个子类需要大量的置换掉超类的行为，那么这个类就不应该是这个超类的子类。

错误地使用继承而不是合成/聚合的一个常见原因是错误地把“Has-A”当成了“Is-A”。“Is-A”代表一个类是另外一个类的一种，而“Has-A”代表一个类是另外一个类的一个角色，而不是另外一个类的特殊种类。

综合上面所陈述的六点原则，我们可以这么来总结，六项原则的关系如图4-4所示。

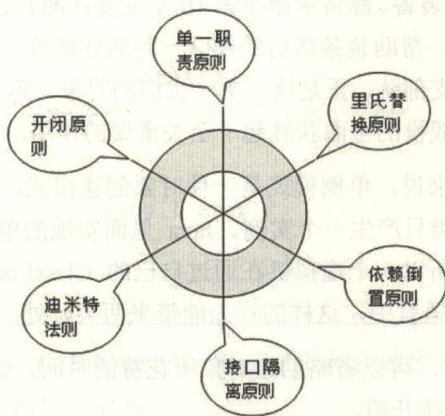


图4-4 六项原则合并

图中的每一条维度各代表一项原则，我们依据对这项原则的遵守程度在维度上画一个点，则如果对这项原则遵守的合理的话，这个点应该落在红色的同心圆内部；如果遵守的差，点将会落在小圆内部；如果过度遵守，点将会落在大圆外部。一个良好的设计体现在图中，应该是六个顶点

都在同心圆中的六边形。

在图 4-5 中, 设计 1、设计 2 属于良好的设计, 他们对六项原则的遵守程度都在合理的范围内; 设计 3、设计 4 设计虽然有些不足, 但也基本可以接受; 设计 5 则严重不足, 对各项原则都没有很好的遵守; 而设计 6 则遵守过度了, 设计 5 和设计 6 都是迫切需要重构的设计。

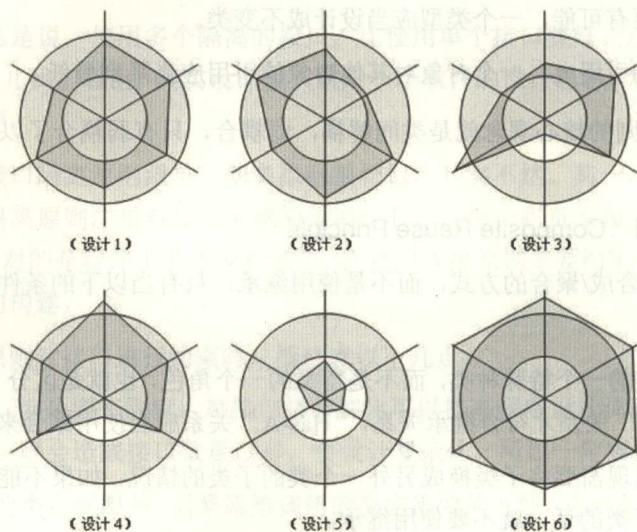


图4-5 六项原则单独分开

4.2.2 单一对象控制

二次世界大战的时候, 我国有一个著名的战役叫“长沙保卫战”, 中国军队指挥官薛岳将军率领第 9 战区十余万将士, 通过所谓的“焦土”战术 4 次瓦解日军的大规模进攻, 给对当时的国民党政府打了一针强心剂。这四次战役中最让人难忘的一幕是, 面对单兵战斗力是中国军队 5 倍的日军, 人数上虽然占据一定优势, 但是只有第 10 军和第 74 军两只军队装备了现代化的军械, 其余军队都是“汉阳造”的落后装备。薛将军命令第 10 军反复在湘北、赣北多处出阵地来回穿插, 面对东西方向出现的多路敌军, 帮助装备落后的部队一起防守阵地, 让敌人误以为是多支部队, 其实薛岳将军只是调动了同一支部队, 正是这一单一实例的对象(第 10 军)在各个战场均发挥出了显著的作用, 为第二次长沙战役的全面获胜起了至关重要的作用。

回到技术主题, 从专业化来说, 单例模式是一种对象创建模式, 它用于产生一个对象的具体实例, 它可以确保系统中一个类只产生一个实例。Java 里面实现的单例是一个虚拟机的范围, 因为装载类的功能是虚拟机的, 所以一个虚拟机在通过自己的 `ClassLoader`⁵ 装载实现单例类的时候就会创建一个类的实例。在 Java 语言中, 这样的行为能带来两大好处:

- (1) 对于频繁使用的对象, 可以省略创建对象所花费的时间, 这对于那些重量级对象而言, 是非常可观的一笔系统开销。
- (2) 由于 `new` 操作的次数减少, 因而对系统内存的使用频率也会降低, 这将减轻 GC 压力, 缩短 GC 停顿时间。

⁵ Java 提供了抽象类 `ClassLoader`, 所有用户自定义类装载器都实例化自 `ClassLoader` 的子类。

因此对于系统的关键组件和被频繁使用的对象，使用单例模式可以有效地改善系统的性能。单例模式的核心在于通过一个接口返回唯一的对象实例。首要的问题就是要把创建实例的权限收回来，让类自身来负责自己类的实例的创建工作，然后由这个类来提供外部可以访问这个类实例的方法，代码如代码清单 4-14 所示。

代码清单 4-14 单例模式基本实现

```
public class Singleton {
    private Singleton() {
        System.out.println("Singleton is create");
    }
    private static Singleton instance = new Singleton();
    public static Singleton getInsatnce() {
        return instance;
    }
}
```

上述代码唯一的不足是无法对 instance 实例做延时加载，例如单例的创建过程很慢，而由于 instance 成员变量是 static 定义的，因此在 JVM 加载单例类时，单例对象就会被建立，如果此时这个单例类在系统中还扮演其他角色，那么在任何使用这个单例类的地方都会初始化这个单例变量，而不管是否会被用到，代码如代码清单 4-15 所示。

代码清单 4-15 单例模式实验

```
public class Singleton {
    private Singleton() {
        System.out.println("Singleton is create");
    }
    private static Singleton instance = new Singleton();
    public static Singleton getInsatnce() {
        return instance;
    }
    public static void createString() {
        System.out.println("createString in Singleton");
    }

    public static void main(String[] args) {
        Singleton.createString();
    }
}
```

可以看到，虽然此时并没有使用单例类，但它还是被创建出来，为了解决这类问题，需要引入延迟加载机制，如代码清单 4-16 所示。

代码清单 4-16 延迟加载的单例模式代码

```
public class LazySingleton {
    private LazySingleton() {
        System.out.println("LazySingleton is create");
    }
}
```

```

private static LazySingleton instance = null;
public static synchronized LazySingleton getInstance(){
    if(instance == null){
        instance = new LazySingleton();
    }
    return instance;
}
public static void createString(){
    System.out.println("create String");
}
public static void main(String[] args){
    LazySingleton.createString();
}
}

```

代码清单 4-16 所示代码首先对于静态成员变量 `instance` 初始化赋值 `null`，确保系统启动时没有额外的负载；其次，在 `getInstance()` 工厂方法中，判断当前单例是否已经存在，若存在则返回，不存在则再建立单例。这里尤其要注意的是，`getInstance()` 方法必须是同步的，否则在多线程环境下，当线程 1 正新建单例时，完成赋值操作前，线程 2 可能判断 `instance` 为 `null`，故线程 2 也将启动新建单例的程序，而导致多个实例被创建，故同步关键字是必需的。由于引入了同步关键字，导致多线程环境下耗时明显增加。

为了解决同步关键字降低系统性能的缺陷，做了一定改进，如代码清单 4-17 所示。

代码清单 4-17 解决同步关键字低效率

```

public class StaticSingleton {
    private StaticSingleton(){
        System.out.println("StaticSingleton is create");
    }
    private static class SingletonHolder{
        private static StaticSingleton instance = new StaticSingleton();
    }
    public static StaticSingleton getInstance(){
        return SingletonHolder.instance;
    }
}

```

代码清单 4-17 的单例模式使用内部类来维护单例的实例，当 `StaticSingleton` 被加载时，其内部类并不会被初始化，故可以确保当 `StaticSingleton` 类被载入 JVM 时，不会初始化单例类，而当 `getInstance()` 方法调用时，才会加载 `SingletonHolder`，从而初始化 `instance`。同时，由于实例的建立是在类加载时完成，故天生对多线程友好，`getInstance()` 方法也无须使用同步关键字。单例模式涉及多线程的相关知识请参见第 5 章。

4.2.3 并程序序设计模式

并程序序设计模式一般有 Future 模式、Master-Slave 模式、保护暂停模式、不变模式、生产者/消费者模式等。

1. Future 模式

Future 模式有点类似商品订单。比如在进行网上购物时,当看中某一件商品时,就可以提交订单。当订单处理完毕后,便可在家里等待商品送货上门。卖家根据订单从仓库里取货,并配送到客户手上。在大部分情况下,商家对订单的处理并不那么快,有时甚至需要几天时间。而在这段时间内,客户不需要在家里等待,而可以去处理其他事务。

将此例类推到程序设计中,当某一段程序提交了一个请求,期望得到一个答复。但非常不幸的是,服务程序对这个请求的处理可能很慢,比如,这个请求可能是通过互联网、HTTP 或者 Web Service 等并不太高效的方式调用的。在传统的单线程环境下,调用函数是同步的,也就是说它必须等到服务程序返回结果后,才能进行其他处理。而在 Future 模式下,调用方式改为异步,而原先等待返回的时间段,在主调用函数中,则可用于处理其他事务。虽然 call 本身仍然需要很长一段时间来处理程序,但是,服务程序不等数据处理完成便立即返回客户端一个伪造的数据(相当于商品的订单,而不是商品本身),实现了 Future 模式的客户端在拿到这个返回结果后,并不急于对其进行处理,而去调用了其他业务逻辑,充分利用了等待时间,这就是 Future 模式的核心所在。在完成了其他业务逻辑的处理后,最后再使用返回比较慢的 Future 数据。这样,在整个调用过程中,就不存在无谓的等待,充分利用了所有的时间片段,从而提高系统的响应速度。

Future 模式的主要参与者包括:

- Main—系统启动,调用 Client 发出请求;
- Client—返回 Data 对象,立即返回 FutureData,并开启 ClientThread 线程装配 RealData;
- Data—返回数据的接口;
- FutureData—Future 数据,构造很快,但是是一个虚拟的数据,需要装配 RealData;
- RealData—真实数据,其构造是比较慢的。

代码清单 4-18 Future 模式示例

```
/*
 * Main 函数主要负责调用 Client 发起请求,并使用返回的数据
 */
public class FutureMain {
    public static void main(String[] args){
        Client client = new Client();
        //这里会立即返回,因为得到的是 FutureData 而不是 RealData
        Data data = client.request("name");
        System.out.println("虚拟请求结束");
        try{
            //这里可以用一个 Sleep 代替对其他业务逻辑的处理,在实际场景中,RealData 被创建,
            //节约了等待时间
            Thread.sleep(1000);
        }catch(InterruptedException ex){
            ex.printStackTrace();
        }
        //使用真实数据
    }
}
```

```
System.out.println(data.getResult);
```

2. Master-Slave 模式

Master-Slave（主从）模式主导的系统架构一般由两类线程实现，Master 线程负责接收和分发任务（将任务拆成一个个子任务），Worker 线程负责处理子任务，每个 Worker 线程只处理部分任务，所有 Worker 线程共同完成所有任务的处理。

该模式的好处在于能够将一个大的任务拆分成若干个小的任务，从而交给不同的 Worker 并行的进行处理，进而提高系统的吞吐量。另外，Client 端一旦提交任务后，Master 线程完成任务的接收和分发后立即返回，因此对客户端来说，整个过程也是异步进行的。

一般的实现思路如下：

- (1) Master 中首先需要维护一个队列 Queue，用于接收任务，同时维护一个所有 Worker 线程的 threadMap，以及每个子任务对应的处理结果集 resultMap，这里由于涉及多线程同时访问 resultMap，因此一般使用 JDK 中的 ConcurrentHashMap 实现；
- (2) Worker 线程实现 Runnable 或继承 Thread，通过 Master 中的 Queue 获取拆分后的子任务，并进行业务处理，并将处理结果设置到 resultMap 中以便 Master 获取到；
- (3) Main 入口函数则负责客户端请求的提交（需要先进程拆解），以及通过 Master 获取各个 Worker 的结果后进行合并，最后返回给客户端完成处理过程。

目前主流的 MapReduce 框架、集群框架很多都是采用该种模式架构实现的。

3. 保护暂停（Guarded Suspension）模式

所谓“保护暂停”模式，核心思想在于仅当服务进程准备好时，才提供服务。它的好处在于既能保证所有的客户端请求均不丢失，同时也避免了服务器由于同时处理太多的请求而崩溃的现象，有效降低系统的瞬时负载，有助于系统稳定性。

其实这种通过中间加一层 Queue 做缓冲的模式在工作中用的很多，类似“ClientThread -> Request Queue -> ServerThread”的情况比比皆是，只不过可能实际中我们往往会结合其他方法一起使用，例如：

- (1) 将 ClientThread 和 ServerThread 均为多个，则变为经典的“生产者-消费者”模式；
- (2) 如果将 ServerThread 拆为 1 个 Master 和多个 Worker，则又是上面提到的“Master-Worker”模式；
- (3) 如果处理的请求需要返回结果，那么又需要和 FutureTask 结合起来使用（即客户端的请求中需要带上 FutureData，并在 ServerThread 中为 FutureData 设置上 RealData）。

4. 不变模式

并发多线程程序中，当多线程对同一个对象进行读写操作时，为了确保对象数据的一致性和准确性，必须进行同步操作，而这正是对系统性能损失严重的地方。因此，为了提高并发程序的

性能，我们可以创建一种不可改变的对象，使用过程中保持不变性。这就是所谓“不变”模式。Java 中这种模式用的很广，如 String、Boolean、Short、Integer、Long、Byte 等。它的好处在于通过回避问题而不是解决问题的态度来处理多线程并发访问控制，但缺点是只适用于对象创建后内部状态和数据不可发生变化的情况。

Java 中不变模式的实现很简单，按照 OO 的思想⁶，只需要满足以下几点即可：

- (1) 将对象的所有属性设为 private final 的；
- (2) 通过 final 修饰 class 确保类不可被继承；
- (3) 去掉对象中的所有 setXX 方法；
- (4) 有包含所有属性的构造函数用于创建对象。

5. 生产者-消费者模式

生产者线程向内存缓冲区提交任务，消费者线程从内存缓冲区获取任务并进行处理。它的好处在于将生产者线程和消费者线程进行解耦，优化系统整体结构，缓解性能瓶颈对系统性能的影响。

Java 中，一般来说使用 LinkedBlockingQueue 作为上面说的“内存缓冲区”，它是阻塞型 BlockingQueue 的一种使用 Link List 的实现，它对头和尾采用两把不同的锁，与 ArrayBlockingQueue 相比提高了吞吐量，适合于实现“生产者-消费者”模式。实现的大致思路如下：

- (1) 创建 Producer 类，实现 run 方法用于提交任务；
- (2) 创建 Consumer 类，实现 run 方法用于处理任务；
- (3) Main 函数中建立缓冲区，若干个生产者，若干个消费者，创建线程池并开始使这些线程工作起来。

关于这一部分内容，我们在第 5 章会有详细的解释。

4.2.4 接口适配

中国古代最初盛行道教，后世逐渐传播的佛教由唐代著名高僧、法相宗创始人玄奘西渡而得。三藏法师为了探究佛教各派学说分歧，于贞观元年一人西行五万里，历经艰辛到达印度佛教中心那烂陀寺取真经。前后十七年学遍了当时的大小乘各种学说，共带回佛舍利 150 粒、佛像 7 尊、经论 657 部，并长期从事翻译佛经的工作。玄奘及其弟子共译出佛典 75 部、1335 卷。玄奘的译典著作有《大般若经》《心经》《解深密经》《瑜伽师地论》《成唯识论》等。《大唐西域记》十二卷，记述他西游亲身经历的 110 个国家及传闻的 28 个国家的山川、地邑、物产、习俗等。《西游记》即以其取经事迹为原型。玄奘由于其爱国及护持佛法的精神和巨大贡献，被誉为“中华民族的脊梁”，他以无我无人无众生无寿者相，不畏生死的精神，西行取佛经，体现了大乘佛法菩萨，渡化众生的真实事迹。他的足迹遍布印度，影响远至日本、韩国以至全世界。他的思想与精神如今已是 中国、亚洲乃至世界人民的共同财富。正是由于有三藏法师这样的翻译者，中国国内才会逐渐盛行佛教，这也是今天我们这一章的主题“适配器模式”。

⁶ 即面向对象思想。

著名的设计模式“四人帮”(Gang of Four)这样评价适配器模式:

将一个类的接口转换成客户希望的另外一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

适配器模式将一个类的接口适配成用户所期待的。一个适配器通常允许因为接口不兼容而不能一起工作的类能够在一起工作,做法是将类自己的接口包裹在一个已存在的类中。

Adapter 设计模式主要目的组合两个不相干类,常用有两种方法,第一种解决方案是修改各自类的接口。但是如果没有源码,或者不愿意为了一个应用而修改各自的接口,则需要使用 Adapter 适配器,在两种接口之间创建一个混合接口。

如图 4-6 所示是适配器模式的类图。Adapter 适配器设计模式中有 3 个重要角色:被适配者 Adaptee、适配器 Adapter 和目标对象 Target。其中两个现存的想要组合到一起的类分别是被适配者 Adaptee 和目标对象 Target 角色,按照类图所示,我们需要创建一个适配器 Adapter 将其组合在一起。

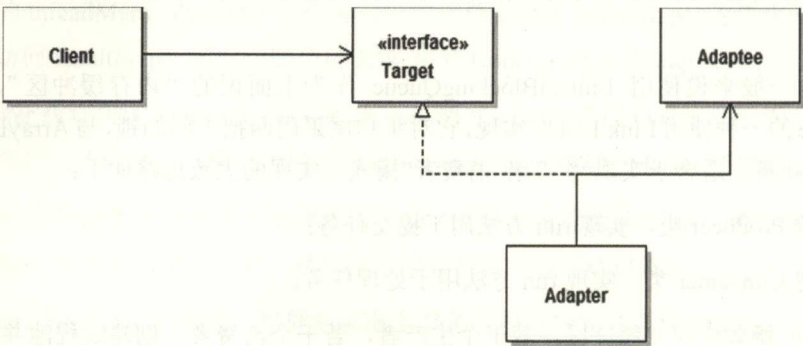


图4-6 适配器模式类图

具体实现代码请见代码清单 4-19。

代码清单 4-19 客户端使用的接口

```
/*
 * 定义客户端使用的接口, 与业务相关
 */
public interface Target {
    /*
     * 客户端请求处理的方法
     */
    public void request();
}
/*
 * 已经存在的接口, 这个接口需要配置
 */
public class Adaptee {
    /*
     * 原本存在的方法
     */
}
```



```

public void specificRequest(){
    //业务代码
}
}

```

代码清单 4-20 适配器实现

```

/*
 * 适配器类
 */
public class Adapter implements Target{
    /*
     * 持有需要被适配的接口对象
     */
    private Adaptee adaptee;
    /*
     * 构造方法，传入需要被适配的对象
     * @param adaptee 需要被适配的对象
     */
    public Adapter(Adaptee adaptee){
        this.adaptee = adaptee;
    }
    @Override
    public void request() {
        // TODO Auto-generated method stub
        adaptee.specificRequest();
    }
}

```

代码清单 4-21 客户端代码

```

/*
 * 使用适配器的客户端
 */
public class Client {
    public static void main(String[] args){
        //创建需要被适配的对象
        Adaptee adaptee = new Adaptee();
        //创建客户端需要调用的接口对象
        Target target = new Adapter(adaptee);
        //请求处理
        target.request();
    }
}

```

以下情况比较适合使用 Adapter 模式：

- (1) 当你想使用一个已经存在的类，而它的接口不符合你的需求；
- (2) 你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类协同工作；

- (3) 你想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口，对象适配器可以适配它的父亲接口。

适配器模式使用示例代码

考虑一个记录日志的应用，用户可能会提出要求采用文件的方式存储日志，也可能会提出存储日志到数据库的需求，这样我们可以采用适配器模式对旧的日志类进行改造，提供新的支持方式。

首先我们需要一个简单的日志对象类，如代码清单 4-22 所示。

代码清单 4-22 日志对象类

```
/*
 * 日志数据对象
 */
public class LogBean {
    private String logId;//日志编号
    private String opeUserId;//操作人员

    public String getLogId(){
        return logId;
    }
    public void setLogId(String logId){
        this.logId = logId;
    }

    public String getOpeUserId(){
        return opeUserId;
    }
    public void setOpeUserId(String opeUserId){
        this.opeUserId = opeUserId;
    }
    public String toString(){
        return "logId="+logId+",opeUserId="+opeUserId;
    }
}
```

接下来定义一个操作日志文件的接口，如代码清单 4-23 所示。

代码清单 4-23 操作日志接口

```
import java.util.List;

/*
 * 读取日志文件，从文件里面获取存储的日志列表对象
 * @return 存储的日志列表对象
 */
public interface LogFileOperateApi {
    public List<LogBean> readLogFile();
    /**
```



```

    * 写日志文件，把日志列表写出到日志文件中
    * @param list 要写到日志文件的日志列表
    */
    public void writeLogFile(List<LogBean> list);
}

```

然后实现日志文件的存储和获取，这里忽略业务代码，如代码清单 4-24 所示。

代码清单 4-24 实现对日志文件的获取

```

import java.io.File;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.List;

/*
 * 实现对日志文件的操作
 */
public class LogFileOperate implements LogFileOperateApi{
    /*
     * 设置日志文件的路径和文件名称
     */
    private String logFileName = "file.log";
    /*
     * 构造方法，传入文件的路径和名称
     */
    public LogFileOperate(String logFilename){
        if(logFilename!=null){
            this.logFileName = logFilename;
        }
    }

    @Override
    public List<LogBean> readLogFile() {
        // TODO Auto-generated method stub
        List<LogBean> list = null;
        ObjectInputStream oin =null;
        //业务代码
        return list;
    }

    @Override
    public void writeLogFile(List<LogBean> list) {
        // TODO Auto-generated method stub
        File file = new File(logFileName);
        ObjectOutputStream oout = null;
        //业务代码
    }
}

```

如果这时候需要引入数据库方式，引入适配器之前，我们需要定义日志管理的操作接口，如代码清单 4-25 所示。

代码清单 4-25 定义数据库操作接口

```
public interface LogDbOpeApi {  
    /*  
     * 新增日志  
     * @param 需要新增的日志对象  
     */  
    public void createLog(LogBean logbean);  
}
```

接下来就要实现适配器了，LogDbOpeApi 接口就相当于 Target 接口，LogFileOperate 就相当于 Adaptee 类。Adapter 类代码如代码清单 4-26 所示。

代码清单 4-26 Adapter 类实现

```
import java.util.List;  
  
/*  
 * 适配器对象，将记录日志到文件的功能适配成数据库功能  
 */  
public class LogAdapter implements LogDbOpeApi {  
    private LogFileOperateApi adaptee;  
    public LogAdapter(LogFileOperateApi adaptee) {  
        this.adaptee = adaptee;  
    }  
    @Override  
    public void createLog(LogBean logbean) {  
        // TODO Auto-generated method stub  
        List<LogBean> list = adaptee.readLogFile();  
        list.add(logbean);  
        adaptee.writeLogFile(list);  
    }  
}
```

最后是客户端代码的实现，如代码清单 4-27 所示。

代码清单 4-27 客户端类实现

```
import java.util.ArrayList;  
import java.util.List;  
  
public class LogClient {  
    public static void main(String[] args) {  
        LogBean logbean = new LogBean();  
        logbean.setLogId("1");  
        logbean.setOpeUserId("michael");  
        List<LogBean> list = new ArrayList<LogBean>();
```



```

LogFileOperateApi logFileApi = new LogFileOperate("");
//创建操作日志的接口对象
LogDbOpeApi api = new LogAdapter(logFileApi);
api.createLog(logbean);
}
}

```

JDK 中有大量使用适配器模式的案例，代码清单 4-28 大致列举了一些类。

代码清单 4-28 使用适配器模式的类

```

java.util.Arrays#asList()
javax.swing.JTable(TableModel)
java.io.InputStreamReader(InputStream)
java.io.OutputStreamWriter(OutputStream)
javax.xml.bind.annotation.adapters.XmlAdapter#marshal()
javax.xml.bind.annotation.adapters.XmlAdapter#unmarshal()

```

JDK1.1 之前提供的容器有 Arrays、Vector、Stack、Hashtable、Properties、BitSet，其中定义了一种访问群集内各元素的标准方式，称为 Enumeration（枚举器）接口，用法如代码清单 4-29 所示。

代码清单 4-29 Enumeration 接口实现方式

```

Vector v=new Vector();
for (Enumeration enum =v.elements(); enum.hasMoreElements();) {
Object o = enum.nextElement();
processObject(o);
}

```

JDK1.2 版本中引入了 Iterator 接口，新版本的集合对象（HashSet、HashMap、WeakHeahMap、ArrayList、TreeSet、TreeMap、LinkedList）是通过 Iterator 接口访问集合元素的，用法如代码清单 4-30 所示。

代码清单 4-30 Iterator 接口实现方式

```

List list=new ArrayList();
for(Iterator it=list.iterator();it.hasNext();)
{
    System.out.println(it.next());
}

```

这样，如果将老版本的程序运行在新的 Java 编译器上就会出错。因为 List 接口中已经没有 elements()，而只有 iterator()了。那么如何可以使老版本的程序运行在新的 Java 编译器上呢？如果不加修改，是肯定不行的，但是修改要遵循“开一闭”原则。我们可以用 Java 设计模式中的适配器模式解决这个问题。代码清单 4-31 所示是解决方法代码。

代码清单 4-31 采用适配器模式

```

import java.util.ArrayList;
import java.util.Enumeration;

```

```

import java.util.Iterator;
import java.util.List;

public class NewEnumeration implements Enumeration
{

    Iterator it;
    public NewEnumeration(Iterator it)
    {
        this.it=it;
        // TODO Auto-generated constructor stub
    }

    public boolean hasMoreElements()
    {
        // TODO Auto-generated method stub
        return it.hasNext();
    }

    public Object nextElement()
    {
        // TODO Auto-generated method stub
        return it.next();
    }

    public static void main(String[] args)
    {
        List list=new ArrayList();
        list.add("a");
        list.add("b");
        list.add("C");
        for(Enumeration e=new NewEnumeration(list.iterator());e.hasMoreElements();)
        {
            System.out.println(e.nextElement());
        }
    }
}

```

代码清单 4-31 所示的 `NewEnumeration` 是一个适配器类，通过它实现了从 `Iterator` 接口到 `Enumeration` 接口的适配，这样我们就可以使用老版本的代码来使用新的集合对象了。

Java I/O 库大量使用了适配器模式，例如 `ByteArrayInputStream` 是一个适配器类，它继承了 `InputStream` 的接口，并且封装了一个 `byte` 数组。换言之，它将一个 `byte` 数组的接口适配成 `InputStream` 流处理器的接口。

我们知道 Java 语言支持四种类型：Java 接口、Java 类、Java 数组、原始类型（即 `int`、`float` 等）。前三种是引用类型，类和数组的实例是对象，原始类型的值不是对象。也即，Java 语言的数组是像所有的其他对象一样的对象，而不管数组中所存储的元素类型是什么。这样一来的话，`ByteArrayInputStream` 就符合适配器模式的描述，是一个对象形式的适配器类。`FileInputStream` 是

一个适配器类。在 `FileInputStream` 继承了 `InputStream` 类型，同时持有一个对 `FileDescriptor` 的引用。这是将一个 `FileDescriptor` 对象适配成 `InputStream` 类型的对象形式的适配器模式。查看 JDK1.4 的源代码我们可以看到代码清单 4-20 所示的 `FileInputStream` 类的源代码。

代码清单 4-32 `FileInputStream` 类

```
Public class FileInputStream extends InputStream {
    /* File Descriptor - handle to the open file */
    private FileDescriptor fd;
    public FileInputStream(FileDescriptor fdObj) {
        SecurityManager security = System.getSecurityManager();
        if (fdObj == null) {
            throw new NullPointerException();
        }
        if (security != null) {
            security.checkRead(fdObj);}fd = fdObj;
        }
        public FileInputStream(File file) throws FileNotFoundException {
            String name = file.getPath();
            SecurityManager security = System.getSecurityManager();
            if (security != null) {
                security.checkRead(name);
            }
            fd = new FileDescriptor();
            open(name);
        }
        //其他代码
    }
}
```

同样，在 `OutputStream` 类型中，所有的原始流处理器都是适配器类。`ByteArrayOutputStream` 继承了 `OutputStream` 类型，同时持有一个对 `byte` 数组的引用。它一个 `byte` 数组的接口适配成 `OutputStream` 类型的接口，因此也是一个对象形式的适配器模式的应用。

`FileOutputStream` 继承了 `OutputStream` 类型，同时持有一个对 `FileDescriptor` 对象的引用。这是一个将 `FileDescriptor` 接口适配成 `OutputStream` 接口形式的对象形适配器模式。

`Reader` 类型的原始流处理器都是适配器模式的应用。`StringReader` 是一个适配器类，`StringReader` 类继承了 `Reader` 类型，持有一个对 `String` 对象的引用。它将 `String` 的接口适配成 `Reader` 类型的接口。

Spring 中使用适配器模式的典型应用

在 Spring 的 AOP 里通过使用的 Advice (通知) 来增强被代理类的功能。Spring 实现这一 AOP 功能的原理就使用代理模式① JDK 动态代理。② CGLib 字节码生成技术代理。) 对类进行方法级别的切面增强，即，生成被代理类的代理类，并在代理类的方法前，设置拦截器，通过执行拦截器重的内容增强了代理类的功能，实现的面向切面编程。

Advice (通知) 的类型有：BeforeAdvice、AfterReturningAdvice、ThrowSAdvice 等。每个类型

Advice(通知)都有对应的拦截器, `MethodBeforeAdviceInterceptor`、`AfterReturningAdviceInterceptor`、`ThrowsAdviceInterceptor`。Spring 需要将每个 Advice(通知)都封装成对应的拦截器类型, 返回给容器, 所以需要使用适配器模式对 Advice 进行转换。具体代码如代码清单 4-33 所示。

代码清单 4-33 `MethodBeforeAdvice` 类

```
public interface MethodBeforeAdvice extends BeforeAdvice {
    void before(Method method, Object[] args, Object target) throws Throwable;
}

public interface MethodBeforeAdvice extends BeforeAdvice {
    void before(Method method, Object[] args, Object target) throws Throwable;
}
```

代码清单 4-34 `Adapter` 类接口

```
public interface AdvisorAdapter {
    boolean supportsAdvice(Advice advice);
    MethodInterceptor getInterceptor(Advisor advisor);
}

public interface AdvisorAdapter {

    boolean supportsAdvice(Advice advice);

    MethodInterceptor getInterceptor(Advisor advisor);
}
```

代码清单 4-35 `MethodBeforeAdviceAdapter` 类

```
class MethodBeforeAdviceAdapter implements AdvisorAdapter, Serializable {

    public boolean supportsAdvice(Advice advice) {
        return (advice instanceof MethodBeforeAdvice);
    }

    public MethodInterceptor getInterceptor(Advisor advisor) {
        MethodBeforeAdvice advice = (MethodBeforeAdvice) advisor.getAdvice();
        return new MethodBeforeAdviceInterceptor(advice);
    }
}

class MethodBeforeAdviceAdapter implements AdvisorAdapter, Serializable {

    public boolean supportsAdvice(Advice advice) {
        return (advice instanceof MethodBeforeAdvice);
    }

    public MethodInterceptor getInterceptor(Advisor advisor) {
        MethodBeforeAdvice advice = (MethodBeforeAdvice) advisor.getAdvice();
```



```
return new MethodBeforeAdviceInterceptor(advice);
```

```
}
```

```
}
```

代码清单 4-36 DefaultAdvisorAdapterRegistry 类

```
public class DefaultAdvisorAdapterRegistry implements AdvisorAdapterRegistry,
Serializable {
    private final List<AdvisorAdapter> adapters = new ArrayList<AdvisorAdapter>(3);

    /**
     * Create a new DefaultAdvisorAdapterRegistry, registering well-known adapters.
     */
    public DefaultAdvisorAdapterRegistry() { //这里注册了适配器
        registerAdvisorAdapter(new MethodBeforeAdviceAdapter());
        registerAdvisorAdapter(new AfterReturningAdviceAdapter());
        registerAdvisorAdapter(new ThrowsAdviceAdapter());
    }

    public Advisor wrap(Object adviceObject) throws UnknownAdviceTypeException {
        if (adviceObject instanceof Advisor) {
            return (Advisor) adviceObject;
        }
        if (!(adviceObject instanceof Advice)) {
            throw new UnknownAdviceTypeException(adviceObject);
        }
        Advice advice = (Advice) adviceObject;
        if (advice instanceof MethodInterceptor) {
            // So well-known it doesn't even need an adapter.
            return new DefaultPointcutAdvisor(advice);
        }
        for (AdvisorAdapter adapter : this.adapters) {
            // Check that it is supported.
            if (adapter.supportsAdvice(advice)) { //这里调用了适配器的方法
                return new DefaultPointcutAdvisor(advice);
            }
        }
        throw new UnknownAdviceTypeException(advice);
    }

    public MethodInterceptor[] getInterceptors(Advisor advisor) throws
UnknownAdviceTypeException {
        List<MethodInterceptor> interceptors = new ArrayList<MethodInterceptor>(3);
        Advice advice = advisor.getAdvice();
        if (advice instanceof MethodInterceptor) {
            interceptors.add((MethodInterceptor) advice);
        }
    }
}
```

```

        for (AdvisorAdapter adapter : this.adapters) {
            if (adapter.supportsAdvice(advice)) { //这里调用了适配器的方法
                interceptors.add(adapter.getInterceptor(advisor));
            }
        }
        if (interceptors.isEmpty()) {
            throw new UnknownAdviceTypeException(advisor.getAdvice());
        }
        return interceptors.toArray(new MethodInterceptor[interceptors.size()]);
    }

    public void registerAdvisorAdapter(AdvisorAdapter adapter) {
        this.adapters.add(adapter);
    }
}

```

双向适配器

适配器也可以实现双向的适配，前面所讲的都是把 `Adaptee` 适配成为 `Target`，其实也可以把 `Target` 适配成为 `Adaptee`。也就是说这个适配器可以同时当作 `Target` 和 `Adaptee` 来使用。

代码清单 4-37 TwiceAdapter 类

```

import java.util.List;

/*
 * 双向适配器对象案例
 */
public class TwiceAdapter implements LogDbOpeApi, LogFileOperateApi {

    /*
     * 持有需要被适配的文件存储日志的接口对象
     */
    private LogFileOperateApi fileLog;

    /*
     * 持有需要被适配的 DB 存储日志的接口对象
     */
    private LogDbOpeApi dbLog;

    public TwiceAdapter(LogFileOperateApi fileLog, LogDbOpeApi dbLog) {
        this.fileLog = fileLog;
        this.dbLog = dbLog;
    }

    @Override
    public List<LogBean> readLogFile() {
        // TODO Auto-generated method stub
        return null;
    }
}

```



```
@Override
public void writeLogFile(List<LogBean> list) {
    // TODO Auto-generated method stub
}
```

```
@Override
public void createLog(LogBean logbean) {
    // TODO Auto-generated method stub
    List<LogBean> list = fileLog.readLogFile();
    list.add(logbean);
    fileLog.writeLogFile(list);
}
```

双向适配器同时实现了 Target 和 Adaptee 的接口，使得双向适配器可以在 Target 或 Adaptee 被使用的地方使用，以提供对所有客户的透明性。尤其在两个不同的客户需要用不同的地方查看同一个对象时，适合使用双向适配器。

对象适配器和类适配器

在标准的适配器模式里面，根据适配器的实现方式，把适配器分成对象适配器和类适配器。

对象适配器

依赖于对象的组合，都是采用对象组合的方式，也就是对象适配器实现的方式。

类适配器

采用多重继承对一个接口与另一个接口进行匹配。由于 Java 不支持多重继承，所以到目前为止还没有涉及。但可以通过让适配器去实现 Target 接口的方式来实现。

代码清单 4-38 ClassAdapter 类

```
import java.util.List;

/*
 * 类适配器对象案例
 */
public class ClassAdapter extends LogFileOperate implements LogDbOpeApi{

    public ClassAdapter(String logFilename) {
        super(logFilename);
        // TODO Auto-generated constructor stub
    }

    @Override
    public void createLog(LogBean logbean) {
        // TODO Auto-generated method stub
        List<LogBean> list = this.readLogFile();
        list.add(logbean);
    }
}
```

```
        this.writeLogFile(list);  
    }  
}
```

在实现中，主要是适配器的实现与以前不一样，与对象适配器实现同样的功能相比，类适配器在实现上有所改变：

- (1) 需要继承 `LogFileOperate` 的实现，然后再实现 `LogDbOpeApi` 接口。
- (2) 需要按照继承 `LogFileOperate` 的要求，提供传入文件路径和名称的构造方法。
- (3) 不再需要持有 `LogFileOperate` 的对象，因为适配器本身就是 `LogFileOperate` 对象的子类。
- (4) 以前调用被适配对象的方法的地方，全部修改成调用自己的方法。

类适配器和对象适配器的选择

- (1) 从实现上：类适配器使用对象继承的方式，属于静态的定义方式。对象适配器使用对象组合的方式，属于动态组合的方式。
- (2) 从工作模式上：类适配器直接继承了 `Adaptee`，使得适配器不能和 `Adaptee` 的子类一起工作。对象适配器允许一个 `Adapter` 和多个 `Adaptee`，包括 `Adaptee` 和它所有的子类一起工作。
- (3) 从定义角度：类适配器可以重定义 `Adaptee` 的部分行为，相当于子类覆盖父类的部分实现方法。对象适配器要重定义 `Adaptee` 很困难。
- (4) 从开发角度：类适配器仅仅引入了一个对象，并不需要额外的引用来间接得到 `Adaptee`。对象适配器需要额外的引用来间接得到 `Adaptee`。

总的来说，建议使用对象适配器方式。

适配器模式使用前提

- (1) 接口中规定了所有要实现的方法。
- (2) 实现此接口的具体类，只用到了其中的几个方法，而其他的方法都是没有用的。

适配器模式实现方法

- (1) 用一个抽象类实现已有的接口，并实现接口中所规定的所有方法，这些方法的实现可以都是“平庸”实现——空方法；但此类中的方法是具体的方法，而不是抽象方法，否则的话，在具体的子类中仍要实现所有的方法，这就失去了适配器本来的作用。
- (2) 原本要实现接口的子类，只实现 1 中的抽象类即可，并在其内部实现时，只对其感兴趣的方法进行实现。

适配器模式使用注意事项

- (1) 充当适配器角色的类就是：实现已有接口的抽象类。
- (2) 为什么要用抽象类？此类是不要被实例化的。而只充当适配器的角色，也就为其子类提供了一个共同的接口，但其子类又可以将精力只集中在其感兴趣的地方。
- (3) 适配器模式中被适配的接口 `Adaptee` 和适配成为的接口 `Target` 是没有关联的，`Adaptee`

和 Target 中的方法既可以是相同的，也可以是不同的。

- (4) 适配器在适配的时候，可以适配多个 Apaptee，也就是说实现某个新的 Target 的功能的时候，需要调用多个模块的功能，适配多个模块的功能才能满足新接口的要求。
- (5) 适配器有一个潜在的问题，就是被适配的对象不再兼容 Adaptee 的接口，因为适配器只是实现了 Target 的接口。这导致并不是所有 Adaptee 对象可以被使用的地方都能是使用适配器，双向适配器解决了这个问题。

优点

适配器模式也是一种包装模式，它与装饰模式同样具有包装的功能，此外，对象适配器模式还具有委托的意思。总的来说，适配器模式属于补偿模式，专用来在系统后期扩展、修改时使用。

缺点

过多使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。

适配器模式应用场景

在软件开发中，也就是系统的数据和行为都正确，但接口不相符时，我们应该考虑用适配器，目的是使控制范围之外的一个原有对象与某个接口匹配。适配器模式主要应用于希望复用一些现存的类，但是接口又与复用环境要求不一致的情况。比如在需要对早期代码复用一些功能等应用上很有实际价值。适用场景大致包含三类：

- (1) 已经存在的类的接口不符合我们的需求。
- (2) 创建一个可以复用的类，使得该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作。
- (3) 在不对每一个都进行子类化以匹配它们的接口的情况下，使用一些已经存在的子类。

4.2.5 访问方式隔离

东汉末年，大将军何进引董卓入京，想借西北王的军队对抗阉党，无奈自己先被阉党做掉，而后造成巨变，导致诸侯并起，最终形成三国鼎立局面。汉献帝即位后，初平三年（公元 192 年），治中从事毛玠向曹操建议“奉天子以令不臣”，曹操采纳了他的建议，迎接汉献帝来到许昌。汉献帝刘协在许都没有实际的权利，曹操不断地诛除公卿大臣，不断地集军政大权于一身。建安元年八月，曹操进驻洛阳，立刻趁张杨、杨奉兵众在外，赶跑了韩暹，接着做了三件事：杀侍中台崇、尚书冯硕等，谓“讨有罪”；封董承、伏完等，谓“赏有功”；追赐射声校尉沮俊，谓“矜死节”。然后在第九天趁他人尚未来得及反应的情况下，迁帝都许，使皇帝摆脱其他势力的控制。此后，他还加紧步伐剪除异己，提高自己的权势。他首先向最有影响力的三公发难，罢免太尉杨彪、司空张喜；其次诛杀议郎赵彦；再次是发兵征讨杨奉，解除近兵之忧；最后是一方面以天子名义谴责袁绍，打击其气焰，另一方面将大将军让予袁绍，稳定大敌。这就是历史上著名的“挟天子以令诸侯”。汉献帝与曹操的关系，是历史上两位伟大的政治家的联手，稳定了东汉政权，最终平稳交接给曹魏政权，也间接映射了我们本节要讲解的“代理模式”。

代理模式使用代理对象完成用户请求，屏蔽用户对真实对象的访问。现实世界的代理人被授权执行当事人的一些事宜，无须当事人出面，从第三方的角度看，似乎当事人并不存在，因为他只和代理人通信。而事实上代理人是要有当事人的授权，并且在核心问题上还需要请示当事人。

在软件设计中，使用代理模式的意图也很多，比如因为安全原因需要屏蔽客户端直接访问真实对象，或者在远程调用中需要使用代理类处理远程方法调用的技术细节（如 RMI），也可能为了提升系统性能，对真实对象进行封装，从而达到延迟加载的目的。

代理模式角色分为 4 种：

- (1) 主题接口：定义代理类和真实主题의 公共对外方法，也是代理类代理真实主题的方法。
- (2) 真实主题：真正实现业务逻辑的类。
- (3) 代理类：用来代理和封装真实主题。
- (4) Main：客户端，使用代理类和主题接口完成一些工作。

延迟加载

以一个简单的示例来阐述使用代理模式实现延迟加载的方法及其意义。假设某客户端软件有根据用户请求去数据库查询数据的功能。在查询数据前，需要获得数据库连接，软件开启时初始化系统的所有的泪，此时尝试获得数据库连接。当系统有大量的类似操作存在时（比如 XML 解析等），所有这些初始化操作的叠加会使得系统的启动速度变得非常缓慢。为此，使用代理模式的代理类封装对数据库查询中的初始化操作，当系统启动时，初始化这个代理类，而非真实的数据库查询类，而代理类什么都没有做。因此，它的构造是相当迅速的。

在系统启动时，将消耗资源最多的方法都使用代理模式分离，可以加快系统的启动速度，减少用户的等待时间。而在用户真正做查询操作时再由代理类单独去加载真实的数据库查询类，完成用户的请求。这个过程就是使用代理模式实现了延迟加载。

延迟加载的核心思想是：如果当前并没有使用这个组件，则不需要真正地初始化它，使用一个代理对象替代它的原有的位置，只要在真正需要的时候才对它进行加载。使用代理模式的延迟加载是非常有意义的，首先，它可以在时间轴上分散系统压力，尤其在系统启动时，不必完成所有的初始化工作，从而加速启动时间；其次，对很多真实主题而言，在软件启动直到被关闭的整个过程中，可能根本不会被调用，初始化这些数据无疑是一种资源浪费。例如使用代理类封装数据库查询类后，系统的启动过程这个例子。若系统不使用代理模式，则在启动时就要初始化 DBQuery 对象，而是用代理模式后，启动时只需要初始化一个轻量级的对象 DBQueryProxy。

如代码清单 4-39 所示，IDBQuery 是主题接口，定义代理类和真实类需要对外提供的服务，定义了视线数据库查询的公共方法 request()函数。DBQuery 是真实主题，负责实际的业务操作，DBQueryProxy 是 DBQuery 的代理类。

代码清单 4-39 延迟加载代理

```
public interface IDBQuery {  
    String request();  
}  
  
public class DBQuery implements IDBQuery{
```



```

public DBQuery(){
    try{
        Thread.sleep(1000); //假设数据库连接等耗时操作
    }catch(InterruptedException ex){
        ex.printStackTrace();
    }
}

@Override
public String request(){
    // TODO Auto-generated method stub
    return "request string";
}
}

```

```

public class DBQueryProxy implements IDBQuery{
    private DBQuery real = null;

    @Override
    public String request() {
        // TODO Auto-generated method stub
        //在真正需要的时候才能创建真实对象，创建过程可能很慢
        if(real==null){
            real = new DBQuery();
        } //在多线程环境下，这里返回一个虚假类，类似于 Future 模式
        return real.request();
    }
}

```

```

public class Main {
    public static void main(String[] args){
        IDBQuery q = new DBQueryProxy(); //使用代理
        q.request(); //在真正使用时才创建真实对象
    }
}

```

动态代理

动态代理是指在运行时动态生成代理类。即代理类的字节码将在运行时生成并载入当前运行的 ClassLoader。与静态处理类相比，动态类有诸多好处。首先，不需要为真实主题写一个形式上完全一样的封装类，假如主题接口中的方法很多，为每一个接口写一个代理方法也很麻烦。如果接口有变动，则真实主题和代理类都要修改，不利于系统维护；其次，使用一些动态代理的生成方法甚至可以在运行时制定代理类的执行逻辑，从而大大提升系统的灵活性。

动态代理类使用字节码动态生成加载技术，在运行时生成加载类。生成动态代理类的方法很

多，如，JDK 自带的动态处理、CGLIB、Javassist 或者 ASM 库。JDK 的动态代理使用简单，它内置在 JDK 中，因此不需要引入第三方 Jar 包，但相对功能比较弱。CGLIB 和 Javassist 都是高级的字节码生成库，总体性能比 JDK 自带的动态代理好，而且功能十分强大。ASM 是低级的字节码生成工具，使用 ASM 已经近乎于在使用 Java bytecode 编程，对开发人员要求最高，当然，也是性能最好的一种动态代理生成工具。但 ASM 的使用很烦琐，而且性能也没有数量级的提升，与 CGLIB 等高级字节码生成工具相比，ASM 程序的维护性较差，如果不是在对性能有苛刻要求的场合，还是推荐 CGLIB 或者 Javassist。

以代码清单 4-39 所示代码中的 DBQueryProxy 为例，使用动态代理生成动态类，替换上例中的 DBQueryProxy。首先，使用 JDK 的动态代理生成代理对象。JDK 的动态代理需要实现一个处理方法调用的 Handler，用于实现代理方法的内部逻辑。

代码清单 4-40 动态代理

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class DBQueryHandler implements InvocationHandler{
    IDBQuery realQuery = null;//定义主题接口

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        // TODO Auto-generated method stub
        //如果第一次调用，生成真实主题
        if(realQuery == null){
            realQuery = new DBQuery();
        }
        //返回真实主题完成实际的操作
        return realQuery.request();
    }
}
```

以上代码实现了一个 Handler，可以看到，它的内部逻辑和 DBQueryProxy 是类似的。在调用真实主题的方法前，先尝试生成真实主题对象。接着，需要使用这个 Handler 生成动态代理对象。如代码清单 4-41 所示。

代码清单 4-41 生成动态代理对象

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class DBQueryHandler implements InvocationHandler{
    IDBQuery realQuery = null;//定义主题接口
```



```

@Override
public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    // TODO Auto-generated method stub
    //如果第一次调用,生成真实主题
    if(realQuery == null){
        realQuery = new DBQuery();
    }
    //返回真实主题完成实际的操作
    return realQuery.request();
}

public static IDBQuery createProxy(){
    IDBQuery proxy = (IDBQuery)Proxy.newProxyInstance(
        ClassLoader.getSystemClassLoader(), new Class[]{IDBQuery.class},
        new DBQueryHandler()
    );
    return proxy;
}
}

```

以上代码生成了一个实现了 IDBQuery 接口的代理类,代理类的内部逻辑由 DBQueryHandler 决定。生成代理类后,由 newProxyInstance() 方法返回该代理类的一个实例。至此,一个完整的动态代理完成了。

在 Java 中,动态代理类的生成主要涉及对 ClassLoader 的使用。以 CGLIB 为例,使用 CGLIB 生成动态代理,首先需要生成 Enhancer 类实例,并指定用于处理代理业务的回调类。在 Enhancer.create() 方法中,会使用 DefaultGeneratorStrategy.Generate() 方法生成动态代理类的字节码,并保存在 byte 数组中。接着使用 ReflectUtils.defineClass() 方法,通过反射,调用 ClassLoader.defineClass() 方法,将字节码装载到 ClassLoader 中,完成类的加载。最后使用 ReflectUtils.newInstance() 方法,通过反射,生成动态类的实例,并返回该实例。基本流程是根据指定的回调类生成 Class 字节码→通过 defineClass() 将字节码定义为类→使用反射机制生成该类的实例。从代码清单 4-42 到代码清单 4-45 所示是使用 CGLIB 动态反射生成类的完整过程。

代码清单 4-42 定义接口

```

public interface BookProxy {
    public void addBook();
}

```

代码清单 4-43 定义实现类

```

//该类并没有申明 BookProxy 接口
public class BookProxyImpl {
    public void addBook() {
        System.out.println("增加图书的普通方法...");
    }
}

```

代码清单 4-44 定义反射类及重载方法

```
import java.lang.reflect.Method;

import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

public class BookProxyLib implements MethodInterceptor {
    private Object target;
    /**
     * 创建代理对象
     *
     * @param target
     * @return
     */
    public Object getInstance(Object target) {
        this.target = target;
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(this.target.getClass());
        // 回调方法
        enhancer.setCallback(this);
        // 创建代理对象
        return enhancer.create();
    }

    @Override
    // 回调方法
    public Object intercept(Object obj, Method method, Object[] args,
        MethodProxy proxy) throws Throwable {
        System.out.println("事物开始");
        proxy.invokeSuper(obj, args);
        System.out.println("事物结束");
        return null;
    }
}
```

代码清单 4-45 运行程序

```
public class TestCglib {
    public static void main(String[] args) {
        BookProxyLib cglib=new BookProxyLib();
        BookProxyImpl bookCglib=(BookProxyImpl)cglib.getInstance(new BookProxyImpl());
        bookCglib.addBook();
    }
}
```

代码清单 4-46 运行输出

事物开始

增加图书的普通方法...

事物结束

代理模式有多种应用场合，如下所述：

- 远程代理，也就是为一个对象在不同的地址空间提供局部代表，这样可以隐藏一个对象存在于不同地址空间的事实。比如说 `WebService`，当我们在应用程序的项目中加入一个 `Web` 引用，引用一个 `WebService`，此时会在项目中声称一个 `WebReference` 的文件夹和一些文件，这个就是起代理作用的，这样可以那个客户端程序调用代理解决远程访问的问题。
- 虚拟代理，是根据需要创建开销很大的对象，通过它来存放实例化需要很长时间的真实对象。这样就可以达到性能的最优化，比如打开一个网页，这个网页里面包含了大量的文字和图片，但我们可以很快看到文字，但是图片却是一张一张地下载后才能看到，那些未打开的图片框，就是通过虚拟代理来替换了真实的图片，此时代理存储了真实图片的路径和尺寸。
- 安全代理，用来控制真实对象访问时的权限。一般用于对象应该有不同访问权限的时候。
- 指针引用，是指当调用真实的对象时，代理处理另外一些事。比如计算真实对象的引用次数，这样当该对象没有引用时，可以自动释放它，或当第一次引用一个持久对象时，将它装入内存，或是在访问一个实际对象前，检查是否已经释放它，以确保其他对象不能改变它。这些都是通过代理在访问一个对象时附加一些内务处理。
- 延迟加载，用代理模式实现延迟加载的一个经典应用就在 `Hibernate` 框架里面。当 `Hibernate` 加载实体 `bean` 时，并不会一次性将数据库所有的数据都装载。默认情况下，它会采取延迟加载的机制，以提高系统的性能。`Hibernate` 中的延迟加载主要分为属性的延迟加载和关联表的延时加载两类。实现原理是使用代理拦截原有的 `getter` 方法，在真正使用对象数据时才去数据库或者其他第三方组件加载实际的数据，从而提升系统性能。

4.3 I/O 及网络相关优化

4.3.1 I/O 操作优化

在实际的生产环境中，I/O 操作往往会成为性能瓶颈，这是不可否认的事实，也是众多开发人员首要考虑的优化问题。如果在 `Windows` 环境下我们使用阻塞 I/O 模型来编写分布式应用，其维护成本会很高。因为客户端的链接数量直接决定了服务器内存开辟的线程数量乘以 2（包含一个输入线程和一个输出线程），并且这些线程是无法采用线程池优化的，因为线程的执行时间大于其创建和销毁时间。长时间的大量并发线程挂起，不仅 CPU 要做实时任务切换，其整体物理资源都将一步步被蚕食，直至最后程序崩溃。在早期的网络变成中，采用阻塞 I/O 模型来编写分布式应用，唯一能做的性能优化只有采取系统的硬件堆机器方式。在付出高昂的硬件成本开销后，项目的可维护性也很低。而且大多数实际项目都是基于 `Linux` 的，跟 `Windows` 内核结构不同的是，`Linux` 环境下是没有真正意义上的线程概念的。其所谓的线程都采用进行模拟的方式，也就是伪线程，因此，对于兵法要求极高的场景下，一旦采用阻塞 I/O 模型无疑是非常不可取的。对有可能存在大量网络 I/O 的应用程序来说，降低系统态 CPU 使用的一个策略是使用 `Java NIO` 的非阻塞数据结构。

Java I/O 模型由同步 I/O 和异步 I/O 构成。同步 I/O 模型包含阻塞 I/O 和非阻塞 I/O，而在 Windows 环境下只要调用 IOCP 的 I/O 模型，就是真正意义上的异步 I/O。IOCP (Input/Output Completion Port, 输入 / 输出完成端口) 简单来说是一种系统级别的高性能异步 I/O 模型。应用程序中所有的 I/O 操作将全部委托给操作系统去执行，直至最后通知并返回结果。Java7 对 IOCP 进行了深度封装，这使得开发人员可以使用 IOCP API 编写高效的分布式应用。注意，由于 IOCP 仅限于使用在 Windows 平台上，因而自然无法再 Linux 平台上使用它，Linux 平台可以使用 Epoll。Epoll 并不是真正意义上的异步 I/O，按照 Unix 网络编程的划分，多路复用 I/O 仍然属于同步 I/O 模型，也就是说 Epoll 其实就是多路复用 I/O。

同步 I/O 方式导致所有的客户端连接在请求服务端时都会阻塞住，等待前面的完成。即使是使用短连接，数据在写入 `OutputStream` 或者从 `InputStream` 读取时都有可能会阻塞。这在大规模的访问量或者系统对性能有要求的时候是不能接受的。具体示例如代码清单 4-47 所示。

代码清单 4-47 I/O 阻塞式示例

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

public class BlockingIODemo {

    public void blockingIoSocket() throws Exception
    {
        ServerSocket serverSocket = new ServerSocket(10002);
        Socket socket = null;
        try
        {
            while (true)
            {
                socket = serverSocket.accept();
                System.out.println("socket 连接:" + socket.getRemoteSocketAddress().
toString());
                BufferedReader in = new BufferedReader(new InputStreamReader(socket.
getInputStream()));
                while(true)
                {
                    String readLine = in.readLine();
                    System.out.println("收到消息" + readLine);
                    if ("end".equals(readLine))
                    {
                        break;
                    }
                }
                //客户端断开连接
            }
        }
    }
}
```



```

        socket.sendUrgentData(0xFF);
    }
}
catch (SocketException se)
{
    System.out.println("客户端断开连接");
}
catch (IOException e)
{
    e.printStackTrace();
}
finally
{
    System.out.println("socket 关闭: " + socket.getRemoteSocketAddress().
toString());
    socket.close();
}
}
}

```

解决这个问题的方法是使用多线程技术，一个客户端一个处理线程，出现阻塞时只是一个线程阻塞而不会影响其他线程工作；为了减少系统线程的开销，采用线程池的办法来减少线程创建和回收的成本。

简单的实现例子如下，使用一个线程（Acceptor）接收客户端请求，为每个客户端新建线程进行处理（Processor），示例代码如代码清单 4-48 所示。

代码清单 4-48 多线程 I/O 阻塞式示例

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;
import java.util.Scanner;

public class MultiThreadBlockingIODemo
{
    public void testMultithreadJIoSocket() throws Exception
    {
        ServerSocket serverSocket = new ServerSocket(10002);
        Thread thread = new Thread(new Acceptor(serverSocket));
        thread.start();

        Scanner scanner = new Scanner(System.in);
        scanner.next();
    }
}

```

```
public class Acceptor implements Runnable
{
    private ServerSocket serverSocket;

    public Acceptor(ServerSocket serverSocket)
    {
        this.serverSocket = serverSocket;
    }

    public void run()
    {
        while (true)
        {
            Socket socket = null;
            try
            {
                socket = serverSocket.accept();
                if(socket != null)
                {
                    System.out.println("收到了 socket: " + socket.getRemoteSocketAddress().
toString());
                    Thread thread = new Thread(new Processor(socket));
                    thread.start();
                }
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        }
    }
}

public class Processor implements Runnable
{
    private Socket socket;

    public Processor(Socket socket)
    {
        this.socket = socket;
    }

    @Override
    public void run()
    {
        try
        {

```



```

        BufferedReader in = new BufferedReader(new InputStreamReader(socket.
getInputStream()));
        String readLine;
        while(true)
        {
            readLine = in.readLine();
            System.out.println("收到消息" + readLine);
            if("end".equals(readLine))
            {
                break;
            }
            //客户端断开连接
            socket.sendUrgentData(0xFF);
            Thread.sleep(5000);
        }
    }
    catch (InterruptedException e)
    {
        e.printStackTrace();
    }
    catch (SocketException se)
    {
        System.out.println("客户端断开连接");
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
    finally {
        try
        {
            socket.close();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
}
}

```

这种阻塞 I/O 的解决方案在大部分情况下是适用的，在出现 NIO 之前是最通常的解决方案，Tomcat 里阻塞 I/O 的实现就是这种方式。但是如果是大量的长连接请求呢？不可能创建几百万个线程保持连接。再退一步，就算线程数不是问题，如果这些线程都需要访问服务端的某些竞争资源，势必需要进行同步操作，这本身就是得不偿失的。

NIO 并不等同于非阻塞 I/O，只要设置 **Blocking** 属性就可以控制阻塞非阻塞。非阻塞 I/O 图。

代码清单 4-49 非阻塞 I/O 方式示例

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;

public class NonBlockingIODemo
{
    Selector selector;
    private ByteBuffer receivebuffer = ByteBuffer.allocate(1024);

    public void testNioNonBlockingSelector()
        throws Exception
    {
        selector = Selector.open();
        SocketAddress address = new InetSocketAddress(10002);
        ServerSocketChannel channel = ServerSocketChannel.open();
        channel.socket().bind(address);
        channel.configureBlocking(false);
        channel.register(selector, SelectionKey.OP_ACCEPT);

        while(true)
        {
            selector.select();
            Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();
            while (iterator.hasNext()) {
                SelectionKey selectionKey = iterator.next();
                iterator.remove();
                handleKey(selectionKey);
            }
        }
    }

    private void handleKey(SelectionKey selectionKey) throws IOException
    {
        ServerSocketChannel server = null;
        SocketChannel client = null;
        if(selectionKey.isAcceptable())
        {
            server = (ServerSocketChannel)selectionKey.channel();
            client = server.accept();
        }
    }
}
```



```

        System.out.println("客户端: " + client.socket().getRemoteSocketAddress().
toString());
        client.configureBlocking(false);
        client.register(selector, SelectionKey.OP_READ);
    }
    if(selectionKey.isReadable())
    {
        client = (SocketChannel)selectionKey.channel();
        receivebuffer.clear();
        int count = client.read(receivebuffer);
        if (count > 0) {
            String receiveText = new String( receivebuffer.array(),0,count);
            System.out.println("服务器端接受客户端数据--:" + receiveText);
            client.register(selector, SelectionKey.OP_READ);
        }
    }
}
}
}

```

Java NIO 提供的非阻塞 I/O 并不是单纯的非阻塞 I/O 模式，而是建立在 Reactor 模式上的 I/O 复用模型；在 I/O multiplexing Model 中，对于每一个 socket，一般都设置成为 non-blocking，但是整个用户进程其实是一直被阻塞的。只不过进程是被 select 这个函数阻塞，而不是被 socket I/O 给阻塞，所以还是属于非阻塞的 I/O。

总的来说，非阻塞 I/O 和阻塞 I/O 最大的不同在于，非阻塞 I/O 并不会在 I/O 请求时产生阻塞，也就是说如果服务端没有收到 I/O 请求时，非阻塞 I/O 会持续轮询方式对 I/O 请求，当有请求进来后开发执行 I/O 操作并阻塞请求进程。Java7 允许开发人员使用 IOCP API 进行异步 I/O 编程，这使得开发人员不必再关心 I/O 是否阻塞，因为程序中所有的 I/O 操作将全部委托给操作系统线程去执行，直至最后通知并返回结果。

4.3.2 Socket 编程

Socket 编程离不开 TCP/IP、UDP 等协议，我们逐一介绍这两个协议。

TCP/IP (Transmission Control Protocol/Internet Protocol)，即传输控制协议/网间协议。TCP/IP 是一个工业标准的协议集，它是为广域网 (WANs) 设计的。举一个例子，如果你要打电话给一个朋友，通常的做法是先拨号，朋友听到电话铃声后提起电话，这时你和你的朋友就建立起了连接，就可以讲话了。等交流结束，挂断电话结束此次交谈。整个流程图如图 4-7 所示。

UDP (User Data Protocol)，即用户数据报协议。UDP 是与 TCP 相对应的协议，它属于 TCP/IP 协议族中的一种。UDP 流程图如图 4-8 所示。

Socket 是应用层与 TCP/IP 协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket 其实就是一个门面模式，它把复杂的 TCP/IP 协议族隐藏在 Socket 接口后面，对用户来说，一组简单的接口就是全部，让 Socket 去组织数据，把具体符合指定协议的处理方法隐藏在 Socket 后面。

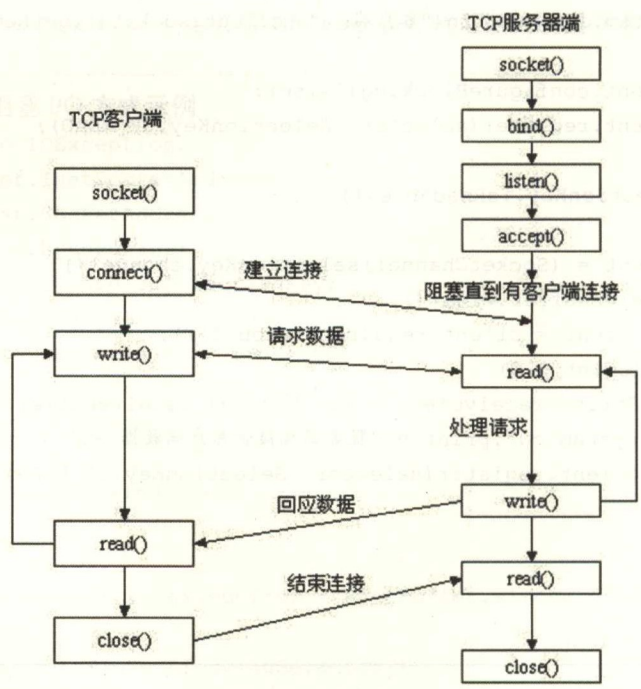


图4-7 TCP/IP流程图

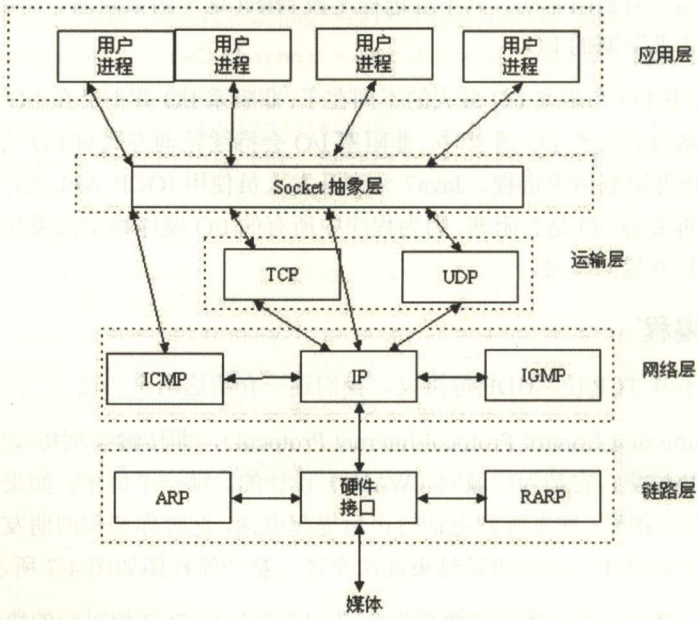


图4-8 UDP流程图

假如我们想使用 Socket，服务器端需要先初始化 Socket，然后与端口绑定 (bind)，对端口进行监听(listen)，接下来调用 accept 阻塞，等待客户端连接。这时如果有个客户端初始化一个 Socket，然后连接服务器 (connect)，如果连接成功，这时客户端与服务器端的连接就建立了。客户端发送数据请求，服务器端接收请求并处理请求，然后把回应数据发送给客户端，客户端读取数据，最后关闭连接，这样一次交互就结束了。

Java 提供的 Socket 包有很多参数，每个参数都有自己的意义。

backlog: 用于 ServerSocket，配置 ServerSocket 的最大客户端等待队列。先看代码清单 4-50 所示的 Socket 示例代码。

代码清单 4-50 Socket 示例代码

```
public class Main {
    public static void main(String[] args) throws Exception {
        int port = 8999;
        int backlog = 2;
        ServerSocket serverSocket = new ServerSocket(port, backlog);
        Socket clientSock = serverSocket.accept();
        System.out.println("revcive from " + clientSock.getPort());
        while (true) {
            byte buf[] = new byte[1024];
            int len = clientSock.getInputStream().read(buf);
            System.out.println(new String(buf, 0, len));
        }
    }
}
```

这段测试代码在第一次处理一个客户端时，就不会处理第二个客户端，所以除了第一个客户端，其他客户端就是等待队列了。所以这个服务器最多可以同时连接 3 个客户端，其中 2 个等待队列。大家可以 telnet localhost 8999 测试下。这个参数设置为 -1 表示无限制，默认是 50 个最大等待队列，如果设置无限制，那么你要小心了，如果你服务器无法处理那么多连接，那么当很多客户端连到你的服务器时，每一个 TCP 连接都会占用服务器的内存，最后会让服务器崩溃的。

另外，就算你设置了 backlog 为 10，如果你的代码中是一直 Socket clientSock = serverSocket.accept()，假设我们的机器最多可以同时处理 100 个请求，总共有 100 个线程在运行，然后你把在 100 个线程的线程池处理 clientSock，不能处理的 clientSock 就排队，最后 clientSock 越来越多，也意味着 TCP 连接越来越多，也意味着我们的服务器的内存使用越来越高(客户端连接进程，肯定会发送数据过来，数据会保存到服务器端的 TCP 接收缓存区)，最后服务器就宕机了。所以如果你不能处理那么多请求，请不要循环无限制地调用 serverSocket.accept()，否则 backlog 也无法生效。如果真的请求过多，只会让你的服务器宕机（相信很多人都这么写，要注意点）。

TcpNoDelay: 禁用纳格算法，将数据立即发送出去。纳格算法是以减少封包传送量来增进 TCP/IP 网络的效能，当我们调用下面代码，如代码清单 4-51 所示。

代码清单 4-51 纳格算法示例

```
Socket socket = new Socket();
socket.connect(new InetSocketAddress(host, 8000));
InputStream in = socket.getInputStream();
OutputStream out = socket.getOutputStream();
String head = "hello ";
String body = "world\r\n";
out.write(head.getBytes());
out.write(body.getBytes());
```

我们发送了 hello, 当 hello 没有收到 ack 确认 (TCP 是可靠连接, 发送的每一个数据都要收到对方的一个 ack 确认, 否则就要重发) 的时候, 根据纳格算法, world 不会立马发送, 会等待, 要么等到 ack 确认 (最多等 100ms 对方会发过来的), 要么等到 TCP 缓冲区内容 \geq MSS, 很明显这里没有机会, 我们写了 world 后再也没有写数据了, 所以只能等到 hello 的 ack 我们才会发送 world, 除非我们禁用纳格算法, 数据就会立即发送了。

SoLinger: 当我们调用 `socket.close()` 返回时, `socket` 已经 write 的数据未必已经发送到对方了, 例如代码清单 4-52 所示。

代码清单 4-52 SoLinger 示例

```
Socket socket = new Socket();
socket.connect(new InetSocketAddress(host, 8000));
InputStream in = socket.getInputStream();
OutputStream out = socket.getOutputStream();
String head = "hello ";
String body = "world\r\n";
out.write(head.getBytes());
out.write(body.getBytes());
socket.close();
```

这里调用了 `socket.close()` 返回时, hello 和 world 未必已经成功发送到对方了, 如果我们设置了 `linger` 而不小于 0, 如代码清单 4-53 所示。

代码清单 4-53 设置 `linger` 不小于 0

```
bool on = true;
int linger = 100; ....
socket.setSoLinger(boolean on, int linger) .....
socket.close();
```

那么 `close` 会等到发送的数据已经确认了才返回。但是如果对方宕机, 超时, 那么会根据 `linger` 设定的时间返回。

UrgentData 和 OOBInline: TCP 的紧急指针, 一般都不建议使用, 而且不同的 TCP/IP 实现, 也不同, 一般说如果你有紧急数据宁愿再建立一个新的 TCP/IP 连接发送数据, 让对方紧急处理。

SoTimeout: 设置 `socket` 调用 `InputStream` 读数据的超时时间, 以毫秒为单位, 如果超过这个时候, 会抛出 `java.net.SocketTimeoutException`。

KeepAlive: `KeepAlive` 不是说 TCP 的常连接, 当我们作为服务端, 一个客户端连接上来, 如果设置了 `keeplive` 为 `true`, 当对方没有发送任何数据过来, 超过一个时间(看系统内核参数配置), 那么我们这边会发送一个 ack 探测包发到对方, 探测双方的 TCP/IP 连接是否有效(对方可能断点, 断网), 在 Linux 好像这个时间是 75 秒。如果不设置, 那么客户端宕机时, 服务器永远也不知道客户端宕机了, 仍然保存这个失效的连接。

SendBufferSize 和 ReceiveBufferSize: TCP 发送缓存区和接收缓存区, 默认是 8192, 一般情况下足够了, 而且就算你增加了发送缓存区, 对方没有增加它对应的接收缓冲, 那么在 TCP 三握手时, 最后确定的最大发送窗口还是双方最小的那个缓冲区, 就算你无视, 发了更多的数据, 那么

多出来的数据也会被丢弃。除非双方都协商好。

4.3.3 NIO2.0 文件系统

I/O 技术算得上是开发过程中用得最多的技术之一，不局限于 Java 语言，在任何变成语言中 I/O 技术都被广泛运用。因为几乎无时无刻开发人员都在直接或者间接地使用着 I/O 技术来完成各式各样的需求和功能。从技术层面上来说，使用程序读 / 写磁盘中某一个指定的文件数据、与数据库建立会话执行 CRUD 操作，甚至使用 Socket 技术进行网络通信等，这些技术底层都需要 I/O 技术作为支撑。

在 Java 中，当需要读取目标数据源的数据时，则需要开启输入流，反之则开启输出流。在此需要注意，数据源涉及较广，因此不仅仅只是本地磁盘文件才算是数据源，内存和网络等载体都可以被称之为数据源。

Java API 中提供了两种类型的数据流，分别是字节流（以 byte 为单位读 / 写数据）和字符流（以 char 为单位读 / 写数据）。字节流为处理字节数据的输入/输出提供了方便的方法，常用于写二进制数据。而字符流则为处理字符数据的输入/输出提供了方便的方法，由于字符流采用的是 Unicode 编码格式，因此字符流可以完美地支持国际化。并且在某种情况下，使用字符流相对于字节流而言更为高效，因为字节流是以 byte 为单位，而字符流以 char 为单位，字符流不仅支持单字符的读 / 写操作，还支持字符数据的整行读 / 写，这一点是字节流无法做到的。但是这两种数据流各有所长，具体应该使用什么类型的数据流，还需要根据具体的应用场景来决定。

早在 Java4 版本中推出了 NIO（Java New Input/Output，Java 新输入 / 输出）开始，开发人员就已经摆脱了阻塞 I/O，目前较为成熟的第三方 NIO 产品有 Mina、Netty 等。

Java NIO 的非阻塞数据结构能够让应用程序在一次网络 I/O（读 / 写）操作中读写更多的数据。我们知道每次网络 I/O 操作最终都会触发系统调用，导致系统态 CPU 的消耗。与 Java NIO 的阻塞式数据结构或者更传统的 Java SE 阻塞式数据结构（譬如 java.net.Socket）相比，Java NIO 的非阻塞数据结构面临的最大挑战是编程的难度较大。例如，只要不超过操作系统的限制，在 Java NIO 的非阻塞输出操作中，可以随心所欲地写入任意数量的数据，但这需要检查输出操作的返回值以确定你要求写入的数据的确已经被写入了。即只要有数据可读，一次 Java NIO 非阻塞输入操作中可以读取任意数量的数据，但是，你同样需要检查最终读取了多少数据。我们需要实现复杂的程序逻辑，处理读取协议数据单元的一部分或者读取多个协议数据单元的情况。换句话说，一次读操作读到的数据可能不足以构造有意义的协议数据单元或消息。这一问题在阻塞式 I/O 中很容易解决，只需要等待直到获取足够的数据构造完整的协议数据单元或消息就行了。

实际应用中，是否需要转向使用非阻塞 I/O 操作则要取决于应用程序对性能的需求。如果你想要利用非阻塞 Java NIO 带来的性能提升，应该考虑使用通用的 Java NIO 框架，尽量减少代码迁移的代价。目前比较流行的 Java NIO 框架有 Grizzly 和 Apache 的 MINA。

Java7 提出了全新的文件系统 NIO2.0，利用 NIO2.0，开发人员则无须关注 I/O 细节，因为新文件系统中封装有大量的通用操作，便于开发人员更好地关注自身业务。并且在 I/O 模型方面，将支持调用操作系统的 IOCP（Input/Output Completion Port，输入 / 输出完成端口）接口实现真正的异步 I/O，尽可能避免因 I/O 问题导致的系统瓶颈出现。NIO2.0 API 改变了针对文件管理的不便，使得在 java.nio.file 包下使用 Path、Paths、Files、WatchService、FileSystem 等这几个常用类型可以

很好地减少开发人员对文件管理的编码量。

Java7 还为开发人员提供了一套全新的文件系统功能，那就是文件检测。以 Web 容器为例，当项目迭代后并重新部署时，开发人员无须对其进行手动重启，因为 Web 容器一旦检测到文件发生改变后，便会自动去适应这些变化并进行相应的更新操作。Web 容器的热发布功能就是基于文件检测功能，所以说，文件检测功能的出现对于 Java 文件系统来说具有重大意义。

总的来说，对于网络 I/O 有一些基本的处理规则如下：

- (1) 减少交互的次数。比如增加缓存，合并请求。
- (2) 减少传输数据大小。比如压缩后传输、约定合理的数据协议。
- (3) 减少编码。比如提前将字符转化为字节再传输。
- (4) 根据应用场景选择合适的交互方式，同步阻塞、同步非阻塞、异步阻塞、异步非阻塞。

4.4 数据应用优化

数据库在现代软件设计里一直保持着较高的使用率，稍大一点的软件都逃不脱数据存储这一业务逻辑。过去我们都把数据存放在关系型数据库，随着大数据需求的不断增多，相关技术的不断完善，我们越来越需要使用 NoSQL 数据库⁷，这一节以 HBase 为例，介绍如何快速、高效地向列式数据库插入大量数据。

4.4.1 关系型数据库优化

关系数据库访问一直是 Java 平台的重要功能，尤其对 Web 应用开发来说，大多数 Web 应用都是由关系数据库来驱动的。Java 平台的数据库访问方式由 JDBC (Java Data Base Connectivity) 规范来定义。JDBC 规范本身也在不断更新，以适应数据库技术的发展，同时满足开发人员的使用需求。Java7 在数据库访问方面的重要更新是增加了对 JDBC4.1 规范的支持，Java6 支持的仅是 JDBC4.0 规范。相对于 JDBC4.0 来说，JDBC4.1 规范又引入了一些新的特性，这些特性都可以在 Java7 中使用，需要注意的是，不同的数据库实现的 JDBC 驱动对 JDBC4.1 规范的支持程度不相同的。特定的数据库实现有可能暂时还不支持某些新特性。JDBC4.1 规范所更新的特性在 `java.sql` 和 `javax.sql` 包中都有。

与数据库访问相关的各种资源，包括数据库连接、查询语句和结果集等，都可以用 `try-with-resources` 语句来进行管理。通过 `try-with-resources` 语句可以去掉数据库操作时对 `close` 方法的调用，大大降低了代码的复杂性。在 Java7 中，`java.sql` 包中的 `java.sql.Connection`、`java.sql.Statement` 和 `java.sql.ResultSet` 接口都继承了 `java.lang.AutoCloseable` 接口，以支持由 `try-with-resources` 语句来管理。下面代码给出了使用 `try-with-resources` 语句进行数据库操作的示例。可以把对 `Connection`、`Statement` 和 `ResultSet` 的创建都封装在 `try-with-resources` 语句中，这样就不用考虑这些对象的关闭操作。

⁷ NoSQL(NoSQL = Not Only SQL)，意即“不仅仅是 SQL”，是一项全新的数据库革命性运动，早期就有人提出，发展至 2009 年趋势越发高涨。NoSQL 的拥护者们提倡运用非关系型的数据存储，相对于铺天盖地的关系型数据库运用，这一概念无疑是一种全新的思维的注入。较常用的有列式数据库、文档数据库、图形数据库。

代码清单 4-54 try-with-resources 语句示例

```

public void doOperation() throws SQLException{
    try(Connection connection = DriverManager.getConnection(
        "jdbc:derby://localhost/java7book");
        Statement stmt = connection.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * FROM book")){
        while(rs.next()){
            System.out.println(rs.getString("name"));
        }
    }
}

```

大部分关系数据库系统都支持为数据库中包含的表和其他对象创建一个额外的名称空间，即模式（schema）。一个模式中 can 包含表、视图以及其他对象。不同模式中可以有名称相同的表或视图，而同一模式中不允许有名称相同的对象存在。通过 SQL 语句“CREATE SCHEMA”可以创建新的模式。当访问某个模式中包含的表或视图等对象时，需要使用模式名称作为前缀来访问，否则无法找到相应的对象。比如，对于一个名为“DEMO_SCHEMA”的模式中的“author”表，在 SQL 语句中应该使用“DEMO_SCHEMA.author”来引用。如果每次都要加上模式名称作为前缀，那么使用起来比较麻烦。JDBC4.1 为 Connection 接口添加了一对新的方法 getSchema 和 setSchema，用来获取和设置数据库操作时使用的默认模式名称。当通过 setSchema 进行设置之后，在 SQL 语句中就不再需要使用模式名称作为前缀了。如代码清单 4-55 所示，如果希望使用 setSchema 方法，那么应该在从 Connection 接口中创建出来的 Statement 对象被使用之前进行设置，否则可能会造成默认的模式无法生效。

代码清单 4-55 setSchema 方法使用方式

```

public void setSchema() throws SQLException{
    try(Connection connection=DriverManager.getConnection("jdbc:derby://localhost/java7book")){
        connection.setSchema("DEMO_SCHEMA");
        try(Statement stmt = connection.createStatement())
        ResultSet rs = stmt.executeQuery("SELECT * FROM author")){
            while(rs.next()){
                System.out.println(rs.getString("name"));
            }
        }
    }
}

```

在使用数据库连接时会遇到的一个问题是，网络连接出现问题造成数据库连接中断。当数据库连接中断时，可能会造成发出数据库操作命令的线程在较长时间内处于等待状态。具体的等待时间取决于 TCP 连接的超时时间设置。这个超时时间一般会长达几分钟。也就是说，在数据库操作命令发出之后，可能需要等待几分钟才能得知数据库连接已经中断了。对于一个数据库应用来说，几分钟的等待时间过长，为此，JDBC4.1 添加了对数据库连接超过的处理方式，主要是在 Connection 接口中新增了 setNetworkTimeout 和 abort 两个方法。

Connection 接口的 `setNetworkTimeout` 方法用来设置通过此数据库连接进行数据库操作时的超时等待时间。如果远程数据库没有在给定的时间内返回操作结果，那么会认为该连接已经关闭，无法再继续使用，处于等待状态的数据库操作方法则会抛出 `SQLException` 异常来表示这种超时的情况。对一个 Connection 接口的实现对象设置要包括 `Statement` 和 `java.sql.PreparedStatement` 接口的实现对象。通过 `Statement` 和 `java.sql.PreparedStatement` 接口的实现对象执行的查询操作也适用于 Connection 接口的实现对象上设置的超时等待时间。可以多次调用 `setNetworkTimeout` 方法，以对不同的情况设置不同的超时时间。对于同一个 Connection 接口的实现对象，如果预计某些查询操作比较耗时，可以把超时等待时间设置为一个较大的值，等操作完成之后，再恢复一个对大多数操作都适用的较小值。

与 `setNetworkTimeout` 相关的方法 `abort` 用来强制关闭一个数据库连接，当调用一个 Connection 接口的实现对象的 `abort` 方法时，这个数据库连接会被标记为关闭状态，同时与该连接相关的资源都会被释放，另外，当前正在使用该连接的方法会抛出 `SQLException` 异常。从作用上讲，`abort` 方法类似于 Connection 接口中已有的 `close` 方法，但是两者的使用场景不同。`close` 方法一般是由 Connection 接口的使用者来调用的，当一个使用者完成数据库操作之后，可以通过 `close` 方法来关闭此数据库连接。而 `abort` 方法一般由数据库连接的管理者调用，如果发生了前面提到的由于网络问题造成数据库连接中断的情况，连接的管理者在检测到问题发生之后，可以调用 `abort` 方法来强制终止此连接，该连接的使用者也不需要等待数据库操作的完成即可进入到 `SQLException` 异常的处理阶段。

`setNetworkTimeout` 和 `abort` 方法都接收一个 `java.util.concurrent.Executor` 接口的实现对象作为参数。这个 `Executor` 接口的实现对象用来执行方法调用中可能会出现的相关任务。对于 `abort` 方法来说，在终止数据库连接的过程中需要释放相关的资源。释放资源的相关任务由该 `Executor` 接口的实现对象来执行。当 `abort` 方法返回之后，`Executor` 接口的实现对象可能仍在继续执行相关的任务。下面代码给出了 `abort` 方法的使用示例。为了查看在 `abort` 方法调用过程中执行的相关任务，这里使用了一个自定义的 `java.util.concurrent.ThreadPoolExecutor` 类的实现。在任务被执行之前，会在控制台输出相关的提示信息。在运行过程中可以发现，在调用 `abort` 方法之后，有新的任务被添加到 `Executor` 接口的实现对象中执行，见代码清单 4-56。

代码清单 4-56 强制关闭连接

```
public class AbortConnection{
    public void abortConnection() throws SQLException{
        Connection connection = DriverManager.getConnection("jdbc:derby://localhost/
java7book");
        ThreadPoolExecutor executor = new DebugExecutorService(2,10,60,TimeUnit.
SECONDS,new LinkedBlockingQueue<Runnable>());
        connection.abort(executor);
        executor.shutdown();
        try{
            executor.awaitTermination(5,TimeUnit.MINUTES);
        } catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}
```



```

    }

    private static class DebugExecutorService extends ThreadPoolExecutor{
        public DebugExecutorService(int corePoolSize,int maximumPoolSize,long
            keepAliveTime,TimeUnit unit,BlockingQueue<Runnable> workQueue){
            super(corePoolSize,maximumPoolSize,keepAliveTime,unit,workQueue);
        }

        public void beforeExecute(Thread t,Runnable r){
            System.out.println("清理任务:" +r.getClass());
            super.beforeExecute(t,r);
        }
    }
}

```

在 JDBC4.1 之前,当使用一个 Statement 接口的实现对象进行查询,得到表示结果集的 ResultSet 接口的实现对象并完成处理之后,需要显式地通过 close 方法来关闭 ResultSet 和 Statement 接口的实现对象以释放资源。使用 Java7 的 try-with-resources 语句可以简化这个操作,而更好的办法是使用 JDBC4.1 为 Statement 接口添加的自动关闭功能。在调用了 Statement 接口的 closeOnCompletion 方法之后,表明当依赖此 Statement 接口的实现对象的所有结果集都被关闭之后,该 Statement 接口的实现对象也被自动关闭。通过这种方式,开发人员只需显式地关闭结果集即可,不需要考虑 Statement 接口的实现对象本身的关闭问题。这个新特性在 Statement 接口的实现对象作为参数在多个方法中传递时尤为实用。当代码中的多个地方都使用了相同的 Statement 接口的实现对象来打开结果集时,以前需要设计好由哪个方法来关闭该 Statement 接口的实现对象,而使用这个自动关闭功能之后,每个方法只需要关闭该方法内部打开的结果集即可,不需要考虑 Statement 接口的实现对象自身的关闭问题。

javax.sql.RowSet 接口是 JDBC2.0 引入的表格式数据的抽象表示形式。RowSet 接口继承自 ResultSet 接口,并在 ResultSet 接口的基础上添加了属性设置和变化通知相关的功能。RowSet 接口是符合 JavaBeans 组件规范的,可以与其他 JavaBeans 组件协同使用。RowSet 接口的使用者并不需要显式地管理数据库连接,只需要把数据库相关的连接信息以属性的方式设置到 RowSet 接口的实现对象中即可。RowSet 接口的实现会负责处理数据库连接的相关工作。RowSet 接口有 5 个不同的子接口,每个子接口实现所适应的场景不同。每个子接口对应的具体实现类也有不少,分别来自不同的提供者。在 Java7 之前并没有一个规范的方式来管理 RowSet 接口的实现对象的创建,开发人员需要直接根据实现类的类名来创建新的 RowSet 接口的实现对象。这种直接依赖具体类而不是接口的做法,不利于代码的移植和维护。

Java7 对 RowSet 接口的实现对象的创建做了更新,采用了 Java 标准的服务提供者接口(Service Provider Interface, SPI)机制。使用者通过工厂方法来创建具体的 RowSet 接口的实现对象,而工厂对象本身由 RowSet 实现的提供者注册到 Java 平台上。具体的工厂对象的查找和创建是由 javax.sql.rowset.RowSetProvider 类的静态方法来完成,具体的工厂对象则实现 javax.sql.rowset.RowSetFactory 接口。RowSetFactory 接口提供了 5 种不同 RowSet 接口实现的创建方法。下面的代码给出了使用 javax.sql.rowset.JdbcRowSet 实现的示例,从中可以看到 RowSet 接口的基本用法,数据库连接字符串和 SQL 语句都是以属性的方式来进行设置的。

代码清单 4-57 JdbcRowSet 示例

```

public void useRowSet() throws SQLException{
    RowSetFactory rsFactory = RowSetProvider.newFactory();
    try(JdbcRowSet jrs = rsFactory.createJdbcRowSet()){
        jrs.setUrl("jdbc:derby://localhost/java7book");
        jrs.setCommand("SELECT * FROM book");
        jrs.execute();
        jrs.absolute(1);
        jrs.updateString("name","New book");
        jrs.updateRow();
    }
}

```

除了上面介绍的几个比较重要的更新之外，JDBC4.1 还有一些比较小的更新。这些小更新包括，在使用 `ResultSet` 中的 `getObject` 方法时，可以直接把结果类型的 Java 类作为参数传递进去，而不需要在调用之后在进行强制类型转换。比如，之前的调用方式是“(String)rs.getObject(1)”，新的调用方式是“rs.getObject(1,String.class)”；表示数据库中元数据的 `java.sql.DatabaseMetaData` 接口中新增了 `getPseudoColumns` 方法来获取数据库表中包含的伪列（pseudo column）和隐藏列（hidden column）的元数据。伪列指的是不在数据库表结构中定义但可以在查询中使用伪列“ROWNUM”来获取当前行的行号，类似的伪列还有获取当前日期和时间戳“SYSDATE”和“SYSTIMESTAMP”。隐藏列通常用来保存一些与数据库相关的内部信息。`getPseudoColumns` 方法的返回值是 `ResultSet` 接口的实现对象，其中的每一行都包含了查找到的伪列和隐藏列的名称、数据类型、大小等信息。如果通过 `Statement` 接口的 `setQueryTimeout` 方法设置了数据库查询的超时等待时间，那么当操作超时的时候，会抛出更加具体的 `java.sql.SQLTimeoutException` 异常，而不是通用的 `SQLException` 异常。

在使用 JDBC4.1 时，需要检测所增加的新特性是否已经被关系数据库实现支持。不同的关系数据库实现在对 JDBC 规范的支持上可能有所不同。在使用 JDBC4.1 的新特性之前，先查看一下所使用的数据库和驱动的相关信息，以确定对 JDBC4.1 规范的支持程度。

4.4.2 向 HBase 插入大量数据

随着业务数据的不断增加，特别是非结构化数据的不断增长，所以传统的关系型数据库已经不能应付现有的需求，迫切需要使用列示数据库。HBase 数据库是一个基于分布式的、面向列的、主要用于非结构化数据存储用途的开源数据库。其设计思路来源于 Google 的非开源数据库“BigTable”。

我们本节需要解决如何向 HBase 数据库导入数据的问题及优化策略，首先介绍一点 HBase 数据库的基础知识。

HBase 的 Rowkey 是数据行的唯一标识，必须通过它进行数据行访问，目前有三种方式，单行键访问、行键范围访问、全表扫描访问。数据按行键的方式排序存储，依次按位比较，数值较大的排列在后，例如 int 方式的排序：1，10，100，11，12，2，20…，906，…。

ColumnFamily 是“列族”，属于 schema 表，在建表时定义，每个列属于一个列族，列名用列族作为前缀“ColumnFamily: qualifier”，访问控制、磁盘和内存的使用统计都是在列族层面进行的。

Cell 是通过行和列确定的一个存储单元，值以字节码存储，没有类型。

Timestamp 是区分不同版本 Cell 的索引，64 位整型。不同版本的数据按照时间戳倒序排列，最新的数据版本排在最前面。

HBase 在行方向上水平划分成 N 个 Region，每个表一开始只有一个 Region，数据量增多，Region 自动分裂为两个，不同 Region 分布在不同 Server 上，但同一个不会拆分到不同 Server。

Region 按 ColumnFamily 划分成 Store，Store 为最小存储单元，用于保存一个列族的数据，每个 Store 包括内存中的 memstore 和持久化到 disk 上的 HFile。

数据导入到 HBase，我们必须考虑分布式环境下的数据合并问题，而数据合并问题一直是 HBase 的难题，因为数据合并需要频繁执行写操作任务，解决方案是我们可以生成 HBase 的内部数据文件，这样可以做到直接把数据文件加载到 HBase 数据库对应的数据表。这样的做法确实写入 HBase 的速度很快，但是如果合并过程中 HBase 的配置如果不是很正确，可能会造成写操作阻塞。目前我们常用的数据导入方法有 HBase Client 调用方式、MapReduce 任务方式、Bulk Load 工具方式、Sqoop 工具方式这四种。

几种方式都可以通过 HFile 的帮助做到快速数据导入，我们首先在这里先给出生成 HFile 的 Java 代码，后面各个方法内部再按照各自方式插入 HFile 文件到 HBase 数据库。如代码清单 4-58 所示。运行代码后生成的 HFile 文件放着后面要用。

代码清单 4-58 生成 HFile

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.KeyValue;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;
import org.apache.hadoop.hbase.mapreduce.HFileOutputFormat;
import org.apache.hadoop.hbase.mapreduce.KeyValueSortReducer;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

public class generateHFile {
    public static class generateHFileMapper extends Mapper<LongWritable, Text,
        ImmutableBytesWritable, KeyValue> {
        @Override
        protected void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
            String line = value.toString();
            String[] items = line.split(",", -1);
```

```

        ImmutableBytesWritable rowkey = new ImmutableBytesWritable(items[0].
getBytes());
        KeyValue kvProtocol = new KeyValue(items[0].getBytes(), "colfam1".
getBytes(), "colfam1".getBytes(), items[0].getBytes());
        if (null != kvProtocol) {
            context.write(rowkey, kvProtocol);
        }
    }
}

public static void main(String[] args) throws IOException, InterruptedException,
ClassNotFoundException {
    Configuration conf = HBaseConfiguration.create();
    System.out.println("conf="+conf);
    HTable table = new HTable(conf, "testtable1");
    System.out.println("table="+table);
    Job job = new Job(conf, "generateHFile");
    job.setJarByClass(generateHFile.class);
    job.setOutputKeyClass(ImmutableBytesWritable.class);
    job.setOutputValueClass(KeyValue.class);
    job.setMapperClass(generateHFileMapper.class);
    job.setReducerClass(KeyValueSortReducer.class);
    job.setOutputFormatClass(HFileOutputFormat.class); //组织成 HFile 文件
    HFileOutputFormat.configureIncrementalLoad(job, table); //自动对 job 进行配置,
SimpleTotalOrderPartitioner 是需要先对 key 进行整体排序, 然后划分到每个 reduce 中, 保证每一个
reducer 中的的 key 最小最大值区间范围, 是不会有交集的。
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}

```

使用 HBase 的 API 中的 Put 方法是最直接的数据导入方式, 该方式的缺点是当需要将海量数据在规定时间内导入 HBase 中时, 需要消耗较大的 CPU 和网络资源, 所以这个方式适用于数据量较小的应用环境。使用 Put 方法将数据插入 HBase 中的方式, 由于所有的操作均是在一个单独的客户端执行, 所以不会使用到 MapReduce 的 job 概念, 即没有任务的概念, 所有的操作都是逐条插入到数据库中的。大致的流程可以分解为 HBase Client 调用 HTable 类访问到 HMaster 的原数据保存地点, 然后通过找到相应的 Region Server, 并分配具体的 Region, 最后操作到 HFile 这一层级。当连接上 HRegionServer 后, 首先获得锁, 然后调用 HRegion 类对应的 put 命令开始执行数据导入操作, 数据插入后还要写时间戳、写 Hlog, WAL(Write Ahead Log)、Hmemstore。具体实现如代码清单 4-59 所示, 在代码中我们尝试插入了 10 万条数据, 打印出插入过程消耗的时间。

代码清单 4-59 采用 HBase Client 方式代码

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;

```



```

import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;

public class PutDemo {

    public static void main(String[] args) throws IOException {
        //创建 HBase 上下文环境
        Configuration conf = HBaseConfiguration.create();
        System.out.println("conf="+conf);
        int count=0;

        HBaseHelper helper = HBaseHelper.getHelper(conf);
        System.out.println("helper="+helper);
        helper.dropTable("testtable1");
        helper.createTable("testtable1", "colfam1");

        HTable table = new HTable(conf, "testtable1");
        long start = System.currentTimeMillis();
        for(int i=1;i<100000;i++){
            //设置 rowkey 的值
            Put put = new Put(Bytes.toBytes("row"+i));
            // 设置 family:qualifier:value
            put.add(Bytes.toBytes("colfam1"), Bytes.toBytes("qual1"),
                Bytes.toBytes("val1"));
            put.add(Bytes.toBytes("colfam1"), Bytes.toBytes("qual2"),
                Bytes.toBytes("val2"));
            //调用 put 方法, 插入数据到 HBase 数据表 testtable1 里
            table.put(put);
            count++;
            if(count%10000==0){
                System.out.println("Completed 10000 rows insetion");
            }
        }

        System.out.println(System.currentTimeMillis() - start);
    }
}

```

清单 4-60 HBaseHelper 类代码部分相关代码

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.KeyValue;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HBaseAdmin;

```

```

import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

/**
 * Used by the book examples to generate tables and fill them with test data.
 */
public class HBaseHelper {
    //在 Java 代码中, 为了连接到 HBase, 我们首先创建一个配置(Configuration) 对象, 使用该对象创
    建一个 HTable 实例。这个 HTable 对象用于处理所有的客户端 API 调用。
    private Configuration conf = null;
    private HBaseAdmin admin = null;

    protected HBaseHelper(Configuration conf) throws IOException {
        this.conf = conf;
        this.admin = new HBaseAdmin(conf);
    }

    public static HBaseHelper getHelper(Configuration conf) throws IOException {
        return new HBaseHelper(conf);
    }

    public void put(String table, String row, String fam, String qual, long ts,
        String val) throws IOException {
        HTable tbl = new HTable(conf, table);
        Put put = new Put(Bytes.toBytes(row));
        put.add(Bytes.toBytes(fam), Bytes.toBytes(qual), ts,
            Bytes.toBytes(val));
        tbl.put(put);
        tbl.close();
    }

    public void put(String table, String[] rows, String[] fams, String[] quals,
        long[] ts, String[] vals) throws IOException {
        HTable tbl = new HTable(conf, table);
        for (String row : rows) {
            Put put = new Put(Bytes.toBytes(row));
            for (String fam : fams) {
                int v = 0;
                for (String qual : quals) {
                    String val = vals[v < vals.length ? v : vals.length];
                    long t = ts[v < ts.length ? v : ts.length - 1];
                    put.add(Bytes.toBytes(fam), Bytes.toBytes(qual), t,

```



```

        Bytes.toBytes(val));
        v++;
    }
}
tbl.put(put);
tbl.close();
}

public void dump(String table, String[] rows, String[] fams, String[] quals)
throws IOException {
    HTable tbl = new HTable(conf, table);
    List<Get> gets = new ArrayList<Get>();
    for (String row : rows) {
        Get get = new Get(Bytes.toBytes(row));
        get.setMaxVersions();
        if (fams != null) {
            for (String fam : fams) {
                for (String qual : quals) {
                    get.addColumn(Bytes.toBytes(fam), Bytes.toBytes(qual));
                }
            }
        }
        gets.add(get);
    }
    Result[] results = tbl.get(gets);
    for (Result result : results) {
        for (KeyValue kv : result.raw()) {
            System.out.println("KV: " + kv +
                ", Value: " + Bytes.toString(kv.getValue()));
        }
    }
}

public void dropTable(String table) throws IOException {
    if (existsTable(table)) {
        disableTable(table);
        admin.deleteTable(table);
    }
}

public void put(String table, String row, String fam, String qual, long ts,
    String val) throws IOException {
    HTable tbl = new HTable(conf, table);
    Put put = new Put(Bytes.toBytes(row));
    put.add(Bytes.toBytes(fam), Bytes.toBytes(qual), ts,

```

```

        Bytes.toBytes(val));
tbl.put(put);
tbl.close();
}

```

如果需要通过编程来生成数据,那么用 `importtsv` 工具不是很方便,这时候可以使用 MapReduce 向 HBase 导入数据,但海量的数据集会让 MapReduce Job 变得很繁重,若处理不当,则可能使得 MapReduce 的 job 运行时的吞吐量很小。由于 MapReduce 在写 HBase 采用的是 `TableOutputFormat` 方式,这样在写入数据库的时候容易对写入块进行频繁的刷新、分割、合并操作,这些操作都是较为耗费磁盘 I/O 的操作,最终导致 HBase 节点的不稳定性。前面介绍过生成 HFile 的代码,生成 HFile 后,我们可以采用 MapReduce 方式把数据导入到 HBase 数据表里,具体如代码清单 4-61 所示。

清单 4-61 MapReduce 方式导入 HFile 到 HBase 数据表

```

import java.io.IOException;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.NullOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class HBaseImportByMapReduce extends Configured implements Tool {
    static final Log LOG = LogFactory.getLog(HBaseImportByMapReduce.class);
    public static final String JOBNAME = "MapReduceImport";
    public static class Map extends Mapper<LongWritable, Text, NullWritable,
NullWritable>{
        Configuration configuration = null;
        HTable xTable = null;
        static long count = 0;

        @Override
        protected void cleanup(Context context) throws IOException, InterruptedException {
            // TODO Auto-generated method stub
            super.cleanup(context);

```



```

        xTable.flushCommits();
        xTable.close();
    }

```

```

@Override

```

```

    protected void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
        String all[] = value.toString().split("/t");
        Put put = new Put(Bytes.toBytes(all[0]));
        put.add(Bytes.toBytes("colfam1"), Bytes.toBytes("value1"), null);
        xTable.put(put);
        if ((++count % 100) == 0) {
            context.setStatus(count + " DOCUMENTS done!");
            context.progress();
            System.out.println(count + " DOCUMENTS done!");
        }
    }

```

```

@Override

```

```

    protected void setup(Context context) throws IOException, InterruptedException {
        // TODO Auto-generated method stub
        super.setup(context);
        configuration = context.getConfiguration();
        xTable = new HTable(configuration, "testtable2");
        xTable.setAutoFlush(false);
        xTable.setWriteBufferSize(12*1024*1024);
    }

```

```

}

```

```

@Override

```

```

    public int run(String[] args) throws Exception {
        String input = args[0];
        Configuration conf = HBaseConfiguration.create(getConf());
        conf.set("hbase.master", "node1:60000");
        Job job = new Job(conf, JOBNAME);
        job.setJarByClass(HBaseImportByMapReduce.class);
        job.setMapperClass(Map.class);
        job.setNumReduceTasks(0);
        job.setInputFormatClass(TextInputFormat.class);
        TextInputFormat.setInputPaths(job, input);
        job.setOutputFormatClass(NullOutputFormat.class);
        return job.waitForCompletion(true) ? 0 : 1;
    }

```

```

    public static void main(String[] args) throws IOException {

```

```

        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();

```

```

        int res = 1;
        try {
            res = ToolRunner.run(conf, new HBaseImportByMapReduce(), otherArgs);
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.exit(res);
    }
}

```

MapReduce 方式启动任务需要一些时间,如果数据量较大,整个 Map 过程也会消耗较多时间。其实一般来说 MapReduce 方式和后面要介绍的 Bulk Load 方式是配合使用的,MapReduce 负责生成 HFile 文件,Bluk Load 负责导入 HBase。

总的来说,使用 Bluk Load 方式由于利用了 HBase 的数据信息是按照特定格式存储在 HDFS 里的这一特性,直接在 HDFS 中生成持久化的 HFile 数据格式文件,然后完成巨量数据快速入库的操作,配合 MapReduce 完成这样的操作,不占用 Region 资源,不会产生巨量的写入 I/O,所以需要较少的 CPU 和网络资源。Bluk Load 的实现原理是通过一个 MapReduce Job 来实现的,通过 Job 直接生成一个 HBase 的内部 HFile 格式文件,用来形成一个特殊的 HBase 数据表,然后将数据文件加载到运行的集群中。使用 Bulk Load 功能最简单的方式就是使用 importtsv 工具,importtsv 是 HBase 的一个内置工具,目的是从 TSV 文件直接加载内容至 HBase。它通过运行一个 MapReduce Job,将数据从 TSV 文件中直接写入 HBase 的表或者写入一个 HBase 的自有格式数据文件,最后提供 CompleteBulkLoad 函数导入 HBase。

代码清单 4-62 MapReduce 方式导入 HFile 到 HBase 数据表

创建生成文件的文件夹:

```
$HADOOP_HOME/bin/hadoop fs -mkdir /user/hac/output
```

开始导入数据:

```
$HADOOP_HOME/bin/hadoop jar /usr/lib/cdh/hbase/hbase-0.94.15-hdh4.6.0.jar importtsv
-Dimporttsv.bulk.output=/user/hac/output/2-1 -Dimporttsv.columns= HBASE_ROW_KEY,
info:name,info:age,info:phone student /user/hac/input/2-1
```

完成 bulk load 导入

```
$HADOOP_HOME/bin/hadoop jar /usr/lib/cdh/hbase/hbase-0.94.15-hdh4.6.0.jar completebulkload
/user/hac/output/2-1 student
```

completebulkload 工具读取生成的文件,判断它们归属的 Resgion Server 族群,然后访问适当的族群服务器。族群服务器会将 HFile 文件转移进自身存储目录中,并且为客户端建立在线数据。

HBase 说明文档里面记载,Bluk Load 方法分为两个主要步骤。

- (1) 使用 HFileOutputFormat 类通过一个 MapReduce 任务方式生成 HBase 的数据文件,就是英文称为“StoreFiles”的数据文件。由于输出的时候按照 HBase 内部的存储格式来输出数据,所以后面读入 HBase 集群的时候就非常高效了。为了保证高效性,HFileOutputFormat 借助 configureIncrementalLoad 函数,基于当前 Table 的各 Region 边界自动匹配 MapReduce 的分区类 TotalOrderPartitioner,这样每一个输出的 HFile 都会是在一个单独的 Region 里面的。

为了实现这样的设计,所有任务的输出都需要使用 Hadoop 的 `TotalOrderPartitioner` 类去对输出进行分区,按照 `Regions` 的主键范围进行分区。`HFileOutputFormat` 类包含了一个快捷方法,即 `configureIncrementalLoad()`,它自动基于数据表的当前 `region` 间隔生成一个 `TotalOrderPartitioner`。

- (2) 完成数据载入到 HBase。当所有的数据都被用 `HFileOutputFormat` 方式准备好以后,我们可以使用 `completebulkload` 读入到集群。这个命令行工具迭代循环数据文件,对于每一个数据文件迅速找到属于它的 `region`,然后 `Region` 服务器会读入这些 `HFile`。如果在生成文件的过程当中 `region` 被修改了,那 `completebulkload` 工具会自动切分数据文件到新的区域,这个过程需要花费一些时间。如果数据表(此处是 `mytable`)不存在,工具会自动创建该数据表。

我们也可以通过自己编写 `Bluk Load` 代码来完成数据导入工作。代码如代码清单 4-63 所示。

代码清单 4-63 MapReduce 方式导入 HFile 到 HBase 数据表

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles;

public class loadIncrementalHFileToHBase {

    public static void main(String[] args) throws Exception {
        Configuration conf = HBaseConfiguration.create();
        HBaseHelper helper = HBaseHelper.getHelper(conf);
        helper.dropTable("testtable2");
        helper.createTable("testtable2", "colfam1");
        HTable table = new HTable("testtable2");
        LoadIncrementalHFiles loader = new LoadIncrementalHFiles(conf);
        loader.doBulkLoad(new Path(args[0]), table);
    }
}
```

我们需要特别提醒以下两点:

- (1) 一定要记得建 HBase 数据表时,针对 `Region` 进行的预切分,这点非常重要。

`HFileOutputFormat.configureIncrementalLoad` 方法会根据 `Region` 的数量来决定 `Reduce` 的数量以及每个 `Reduce` 覆盖的 `RowKey` 范围,否则单个 `Reduce` 过大,容易造成任务处理不均衡。造成这个的原因是,创建 HBase 表的时候,默认只有一个 `Region`,只有等到这个 `Region` 的大小超过一定的阈值之后,才会进行 `split`,所以为了利用完全分布式加快生成 `HFile` 和导入 HBase 中以及数据负载均衡,我们需要在创建表的时候预先进行分区,而进行分区时要利用 `startKey` 与 `endKey` 进行 `rowKey` 区间划分(因为导入 HBase 中,需要 `rowKey` 整体有序)。解决方法是在数据导入之前,自己先写一个 MapReduce 的 Job 求最小与最大的 `rowKey`,即 `startKey` 与 `endKey`。

(2) 单个 RowKey 下的子列不要过多，否则在 reduce 阶段排序的时候会造成内存溢出异常，有一种办法是通过二次排序来避免 reduce 阶段的排序，这个解决方案需要视具体应用而定。

Sqoop 是 Apache 顶级项目，主要用于在 Hadoop (Hive) 与传统的数据库 (mysql、postgresql 等) 之间进行数据的传递，可以将一个关系型数据库，例如 MySQL、Oracle、Postgres 等中的数据导入到 Hadoop 的 HDFS 中，也可以将 HDFS 的数据导进到关系型数据库中。Sqoop 支持多种导入方式，包括指定列导入、指定格式导入、支持增量导入 (有更新才导入) 等。Sqoop 的一个特点就是可以通过 Hadoop 的 MapReduce 把数据从关系型数据库中导入数据到 HDFS。

Sqoop 的架构较为简单，通过整合 Hive，实现 SQL 方式的操作，通过整合 HBase，可以向 HBase 写入数据，通过整合 Oozie，拥有了任务流的概念。而 Sqoop 本身是通过 MapReduce 机制来保证传输数据，从而提供并发特性和容错机制，系统架构图⁸如图 4-9 所示。

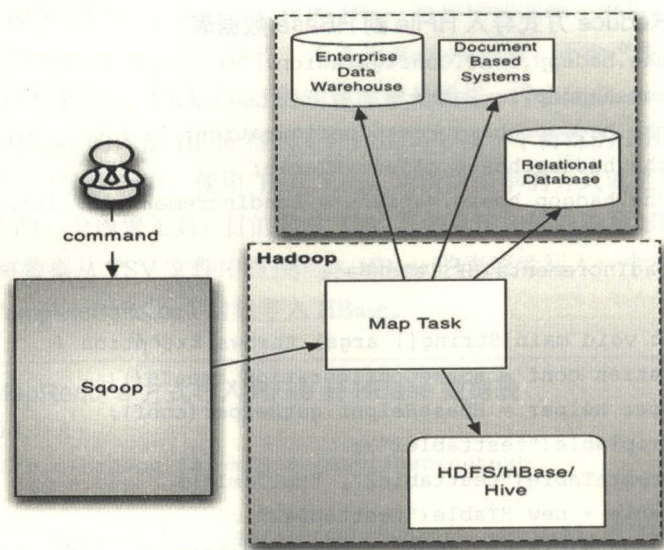


图4-9 Sqoop系统架构图

我们这次做的实验使用了 Sqoop 的 import 功能，用于将 Oracle 中的人员信息导入到 HBase。在 Hadoop 和 HBase 正常运行的环境里，我们首先需要配置好 Sqoop，然后调用如下的命令即可将 Oracle 中的表导入到 HBase 中，代码如代码清单 4-64 所示。

代码清单 4-64 Sqoop 导入 Oracle 数据到 HBase

```
sqoop import
--connect jdbc:oracle:thin:@172.7.27.225:1521:testzmy //JDBC URL
--username SYSTEM //Oracle username (必须大写)
--password hik123456 //Oracle password
--query 'SELECT RYID, HZCZRK_JBXXB.ZPID, HZCZRK_JBXXB.GMSFHM, HZCZRK_JBXXB.XM,
HZCZRK_JBXXB.XB, HZCZRK_JBXXB.CSRQ, HZCZRK_ZPXXB.ZP AS ZP FROM HZCZRK_JBXXB JOIN
```

⁸ 来源 Apache 官方网站。


```

HZCZRK_ZPXXB USING(RYID) WHERE $CONDITIONS'
// Oracle 数据, Sqoop 支持多表 query
--split-by RYID //指定并行处理切分任务的列名, 通常为主键
--map-column-java ZP=String //ZP 为 LONG RAW 类型, sqoop 不支持, 需要映射成 String
--hbase-table TESTHZ //HBase 中的 Table
--column-family INFO //HBase 中的 column-family

```

代码清单 4-64 所示从两张数据表 HZCZRK_JBXXB 和 HZCZRK_ZPXXB 读取数据并写入到 HBase 数据表 TESTHZ, 该数据表有一个列组 INFO。我们在 VMWare CentOS5.6 单节点伪分布式环境下进行了测试。测试结果显示, 单表 HZCZRK_ZPXXB 导入 90962 条数据耗时约 27 分钟, 两表 HZCZRK_JBXXB 和 HZCZRK_ZPXXB JOIN 导入 90962 条数据耗时约 50 分钟。

该实验显示 Sqoop 使用过程中的局限性:

- (1) Import 中进行多表 query 的方式效率会受到影响。
- (2) 不支持从数据库的视图导出数据。
- (3) 不支持 BLOB、RAW 等大数据块类型直接导入到 HBase, 需要通过--map-column-java 将对应的列映射到 java 的基本类型 String 来处理。
- (4) 每次 import 只能导入到 HBase 的一个 column family。

总的来说, Sqoop 类似于其他 ETL 工具, 使用元数据模型来判断数据类型, 并在数据从数据源转移到 Hadoop 时确保类型安全的数据处理。Sqoop 专为大数据批量传输设计, 能够分割数据集并创建 Hadoop 任务来处理每个区块。

4.4.3 解决海量数据缓存

大规模、大数据量、高并发企业级或者互联网应用为了解决数据缓存、降低数据库负载、提高查询性能等突出问题, 很多采用了 Hazelcast、Oracle Coherence、GemFire (比如 12306 网站), 或者目前应用越来越广泛的 Redis 等缓存技术、Apache Ignite 内存数组组织框架等各种技术。

Hazelcast

Hazelcast 是一个高度可扩展的数据分发和集群平台。特性包括:

- 提供 java.util.{Queue, Set, List, Map} 分布式实现;
- 提供 java.util.concurrent.locks.Lock 分布式实现;
- 提供 java.util.concurrent.ExecutorService 分布式实现;
- 提供用于一对多关系的分布式 MultiMap;
- 提供用于发布/订阅的分布式 Topic (主题);
- 通过 JCA 与 J2EE 容器集成和事务支持;
- 提供用于安全集群的 Socket 层加密;
- 支持同步和异步持久化;
- 为 Hibernate 提供二级缓存 Provider;

- 通过 JMX 监控和管理集群;
- 支持动态 HTTP Session 集群;
- 利用备份实现动态分割;
- 支持动态故障恢复。

Oracle Coherence

Coherence 是 Oracle 为了建立一种高可靠和高扩展集群计算的一个关键部件, 集群指的是多于一个应用服务器参与到运算里。Coherence 的主要用途是共享一个应用的对象 (主要是 Java 对象, 比如 Web 应用的一个会话 Java 对象) 和数据 (比如数据库数据, 通过 OR-MAPPING 后成为 Java 对象)。简单来说, 就是当一个应用把它的对象或数据托管给 Coherence 管理的时候, 该对象或数据就能够在整个集群环境 (多个应用服务器节点) 共享, 应用程序可以非常简单地调用 get 方法取得该对象, 并且由于 Coherence 本身的冗余机制使得任何一个应用服务器节点的失败都不会影响到该对象的丢失。其实如果不使用 Coherence, 对于一个会话在多个应用服务器节点的共享一般是通过应用服务器本身的集群技术, 而 Coherence 的创造者则认为基于某种应用服务器技术的集群技术来共享会话变量的技术并不完整, 而专门开发出 Coherence 这个产品 (原来称为 tangosol) 并且最后被 Oracle 收购, 这个产品既有原来各种应用服务器集群所具有的各种技术特点, 而且又增加了原来各种应用服务器集群技术所没有的各种特性。

一般而言, 整个应用架构的扩展性由架构里的最不能扩展的部位 (称之为瓶颈) 决定, 这个瓶颈一般而言都是数据源的处理, Coherence 针对这种理解提供了应用层的数据共享缓冲, 任何一个时候如果应用能够从这个数据缓冲里满足要求, 则不会将请求发给数据源, 从而极大地增强一般的瓶颈 (数据) 的扩展性。

为了加强数据的写处理性能, Coherence 还设计了延迟写的功能, 就是应用的写会先缓存在 Coherence 的缓冲区, 然后延迟写到数据库里, 为了减轻数据源的写压力, Coherence 只把最近的更改写到数据源, 比如一条数据被更改了多遍, 则只有最后的更改会被提交到数据源。而且, 如果可能, 多个 SQL 语句会被变成一个 SQL 语句批, 一次提交给数据源, 这样又极大地降低了对数据源的压力。

Coherence 的第二个非常重要的特地是支持数据的分区处理, 就是如果有 N 个处理节点, 则每个节点只管理 $1/N$ 的数据, 当一个节点失效时, 该节点的数据会在剩下的节点均分, 每个节点将管理 $1/(N-1)$ 的数据。同样的, 当一个节点增加进来时, 则每一个节点都会分配一部分数据给新的节点, 则最终每个节点只管理 $1/(N+1)$ 的数据。大家知道, 一般应用服务器的集群都有只能缓冲共享 2G Java 对象的缺点, 而 Coherence 这种设计让 Coherence 能够处理非常多的数据, 只需要通过增加节点的数量, 就可以处理更多的数据。

如果安装了 Coherence, 则应用服务器不需要配置专有的服务器集群技术, 因为 Coherence*web 模块提供了可用于处理 HTTP 会话信息在 Coherence 集群内共享的功能, 当一个节点需要读取 HTTP 会话信息而发现自己没有该会话信息的时候, 它会把请求同时发给所有的节点 (multicast), 而当一个节点需要写 HTTP 会话信息的同时, 它也会把写请求发给所有的节点, 所以 2 个节点的处理和 100 个节点的处理都是一样的。

GemFire

GemFire 是一个位于应用集群和后端数据源之间的高性能、分布式的操作数据(operational data)管理基础架构。它提供了低延迟、高吞吐量的数据共享和事件分发。GemFire 充分利用网络中的内存和磁盘资源,形成一个实时的数据网格(data fabric or grid)。

在 P2P 分布式系统中,应用程序使用 GemFire 的镜像(mirroring)功能来将大量数据跨结点分区(sharding)以及在这些结点间进行数据复制同步。因为在 P2P 拓扑中缓存数据与应用在一起,所以首先说一下嵌入式缓存。所谓嵌入式缓存(embedded cache)其实就是说缓存和应用程序在一起,直接利用应用服务器的内存空间。也就是我们常说的类似 Ehcache 的那种本地缓存(local cache)。

Mirrored 结点就像一块磁铁一样,将其他数据区域的数据都吸附过来,形成一块完整的数据集合。当一块数据区域被配置为 mirrored 的结点第一次新建或重建时, GemFire 将自动执行初始镜像抓取(initial image fetch)操作,从其他结点的数据子集中还原出完整的状态。如果此时网络中存在另一个 mirrored 结点,那么将会执行最优直接抓取(optimal directed fetch)。

不同于 mirrored 结点,每个 partitioned 结点都持有唯一的一块数据。应用程序就像操作本地数据一样, GemFire 在幕后管理各个分区的数据,并且保证在至多一跳内(at most one network hop)完成数据访问。根据 GemFire 的哈希算法,分区数据会被自动放入到各个结点的 bucket 中。同时 GemFire 也会自动分配出冗余数据的位置并进行复制。当某个结点出错时,客户端请求会自动被重定向到备份结点。并且 GemFire 会重新复制出一份数据,从而保证数据的冗余拷贝数。最后,我们可以随时向网络中加入新的结点来对 GemFire 集群进行动态扩容。

默认 GemFire 使用 IP 多播来发现新成员,然而所有成员间的通信都采用 TCP。对于部署环境禁止使用 IP 多播或者网络跨越多个子网时, GemFire 提供备用方法:使用轻量级的定位服务器(locator server)来追踪所有成员的连接。新成员加入集群时,将询问定位服务并建立类似于 IP 多播的 socket 到 socket 的 TCP 连接。

12306 采用的是 GemFire,根据他们发表的文章,我们可以知道根据系统运行数据记录,技术改造之后,在只采用 10 几台 X86 服务器实现了以前数十台小型机的余票计算和查询能力,单次查询的最长时间从之前的 15 秒左右下降到 0.2 秒以下,缩短了 75 倍以上。2012 年春运的极端高流量并发情况下,支持每秒上万次的并发查询,高峰期间达到 2.6 万 QPS 吞吐量,整个系统效率显著提高。订单查询系统改造,在改造之前的系统运行模式下,每秒只能支持 300-400 个 QPS 的吞吐量,高流量的并发查询只能通过分库来实现。改造之后,可以实现高达上万个 QPS 的吞吐量,而且查询速度可以保障在 20 毫秒左右。新的技术架构可以按需弹性动态扩展,并发量增加时,还可以通过动态增加 X86 服务器来应对,保持毫秒级的响应时间。

Redis

Redis 是一个开源的使用 ANSI C 语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value 数据库,并提供多种语言的 API。

Redis 是一个 key-value 存储系统。和 Memcached 类似,它支持存储的 value 类型相对更多,包括 string(字符串)、list(链表)、set(集合)、zset(sorted set——有序集合)和 hash(哈希类型)。这些数据类型都支持 push/pop、add/remove 及取交集并集和差集及更丰富的操作,而且这些操作都是原子性的。在此基础上,Redis 支持各种不同方式的排序。与 Memcached 一样,为了保

证效率，数据都是缓存在内存中。区别的是 Redis 会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了 master-slave(主从)同步。

在 Redis 中，并不是所有的数据都一直存储在内存中的。这是和 Memcached 相比一个最大的区别。Redis 只会缓存所有的 key 的信息，如果 Redis 发现内存的使用量超过了某一个阈值，将触发 swap 的操作，Redis 根据“ $\text{swappiness} = \text{age} * \log(\text{size_in_memory})$ ”计算出哪些 key 对应的 value 需要 swap 到磁盘。然后再将这些 key 对应的 value 持久化到磁盘中，同时在内存中清除。这种特性使得 Redis 可以保持超过其机器本身内存大小的数据。当然，机器本身的内存必须要能够保持所有的 key，毕竟这些数据是不会进行 swap 操作的。同时由于 Redis 将内存中的数据 swap 到磁盘中的时候，提供服务的主线程和进行 swap 操作的子线程会共享这部分内存，所以如果更新需要 swap 的数据，Redis 将阻塞这个操作，直到子线程完成 swap 操作后才可以进行修改。

Apache Ignite 内存数组组织框架

Apache Ignite 是一个高性能、集成和分布式的内存计算和事务平台，用于大规模的数据集处理，比传统的基于磁盘或闪存的技术具有更高的性能，同时他还为应用和不同的数据源之间提供高性能、分布式内存中数据组织管理的功能。我们需要理解，将数据存储在缓存中能够显著地提高应用的速度，因为缓存能够降低数据在应用和数据库中的传输频率。Ignite 来源于尼基塔·伊万诺夫于 2007 年创建的 GridGain 系统公司开发的 GridGain 软件，尼基塔领导公司开发了领先的分布式内存片内数据处理技术领先的 Java 内存片内计算平台，今天在全世界每 10 秒它就会启动运行一次。

Ignite 和 Hadoop 解决的是不同的问题，即使在一定程度上可能应用了类似的底层基础技术。Ignite 是一种多用途，和 OLAP/ OLTP 内存中数据结构相关的，而 Hadoop 仅仅是 Ignite 原生支持（和加速）的诸多数据来源之一。Spark 是一个和 Ignite 类似的项目。但是 Spark 聚焦于 OLAP，而 Ignite 凭借强大的事务处理能力在混合型的 OLTP/ OLAP 场景中表现更好。特别是针对 Hadoop，Ignite 将为现有的 Map/Reduce, Pig 或 Hive 作业提供即插即用式的加速，避免了推倒重来的做法，而 Spark 需要先做数据 ETL，更适合新写的分析应用。

Apache Ignite 允许用户将常用的热数据储存在内存中，它支持分片和复制两种方式，让开发者可以均匀地将数据分布式到整个集群的主机上。同时，Ignite 还支撑任何底层存储平台，不管是 RDBMS、NoSQL，又或是 HDFS。在集群配置好之后，数据集增加只需在 Ignite 集群中增加节点而不需要重启整个集群。节点数目可以无限增加，所以 Ignite 的扩展性是无穷的。

在 Ignite 的配置上有下面这几个选项可供选择：

- 在 Write-Through 模式中，缓存中的数据更新会被同步更新到数据库中。Read-Through 模式则是指请求的数据在缓存中不可用时，会自动从数据库中拉取。
- Ignite 还提供了一种叫作 Write-Behind Caching 的数据库异步更新模式。默认情况下，Write-Through 中每一次更新都会对数据库发起一次请求。如果使用 Write-Behind Caching 后写，对缓存的更新会整合成批次然后再发送给数据库。这对改删频繁的应用来说可以达到相当的性能提升。

Ignite 提供了易用的 schema 映射工具，从而系统可以自动地与数据库整合。这一工具可以自动地连接数据库，并生成所有需要的 XML OR-mapping 配置以及 Java 域模型 POJOs。

查询 Ignite 缓存很简单，使用的就是标准的 SQL。Ignite 支持所有的 SQL 函数、聚合和 group 操作，甚至支持分布式 SQL JOINS。下面 Ignite 中一个 SQL 查询示例。

代码清单 4-65 Ignite SQL 查询示例

```
IgniteCache<Long, Person> cache = ignite.cache("mycache");
// 'Select' query to concatenate the first and last name of all persons.
SqlFieldsQuery sql = new SqlFieldsQuery(
    "select concat(firstName, ' ', lastName) from Person");
// Execute the query on Ignite cache and print the result.
try (QueryCursor<List<?>> cursor = cache.query(sql)) {
    for (List<?> row : cursor)
        System.out.println("Full name: " + row.get(0));
}
```

Ignite 上也支持 Java8 的 Lambda 方法的使用。在 Ignite 的分布式网格中，创建 GridInstance 实例，grid 通过 Broadcast（广播）的方式来执行 GridRunnable 方法中的操作，如代码清单 4-66 所示。

代码清单 4-66 Ignite 网格编程

```
try(Grid grid = GridGain.start()){
    grid.compute().broadcast((GridRunnable) ()->
        System.out.println("HelloWorld")).get();
}
```

在 Ignite 中使用时空索引，能够轻松地使用带有内存索引的 SQL 语句查询内存中的数据，并且能够扩展 SQL 到时空查询。例如，下面的代码中的查询将找到地图中的特定平方区域内所有的点。

代码清单 4-67 Lambda 方式返回所有的点

```
Polygon square = factory.createPolygon(new Coordinate[]{
    new Coordinate(0,0),
    new Coordinate(0,100),
    new Coordinate(100,100),
    new Coordinate(100,0),
    new Coordinate(0,0)
});

cache.queries().
    createSqlQuery(MapPoint.class,"select * from MapPoint where location && ?").
    queryArguments(square).
    execute().get();
}
```

Apache Ignite 是一个聚焦分布式内存计算的开源项目，它在内存中储存数据，并分布在多个节点上以提供快速数据访问。此外，可选地将数据同步到缓存层同样是一大优势。最后，可以支持任何底层数据库存储同样让 Ignite 成为数据库缓存的首选。

4.5 其他优化

4.5.1 Web 系统性能优化建议

在最近若干年里,由于互联网行业的流行,Web 应用已经变得异常复杂。Web 应用不断地被添加复杂的特性,每天还要处理几百万甚至上千万的请求。为了达到最理想的性能,构建适当的应用架构并对它的运行容器进行调优就显得非常重要。

多个因素都会影响 Web 站点的性能,包括服务器分发页面的时间、网络延迟以及浏览器显示页面的时间。一般来说,糟糕的页面设计会延长页面的显示时间最终导致用户的不满。所以,生成高效的 Web 页面,是 Web 应用设计中最重要步骤之一。

Web 应用的基准测试有几个方面。

- (1) 对于页面访问模式复杂的应用,可以基于马尔科夫链制定基准测试。对于各页面访问彼此无关的应用,可依据每个页面的预期访问量,在基准测试中设定相应的访问比例,以此降低复杂性。
- (2) 读取日志文件可以很好地模拟生产环境负载情况。在生产环境中,Web 服务器接收请求时通常也会记录在访问日志中,可以在基准测试中回放日志,以便尽可能地模拟生产负载。需要特别留意那些更改数据的请求(POST、PUT、DELETE)。基准测试需要某个生产环境的数据副本,然后在此基础上回放数据更改请求,从而保证生产数据的一致性不会被破坏。
- (3) 如果页面有多个 Ajax 请求,衡量包含所有相关请求的页面整体性能就很重要了。
- (4) 如果应用的行为因人而异,会使基准测试的制定变得更为困难。这类应用的例子是社交网络,它分发的内容与请求用户有关。非登录用户的请求通常由缓存分发。这些请求相当于给缓存进行负载测试。即便是登录用户,不同用户的应用逻辑也有很大差别。比如,对于只有少量朋友的用户与有许多朋友的用户,应用性能相差很大。为这类应用制定精确的基准测试模型很困难,因为请求分发除了需要基于页面 URL 之外,还需要基于用户。

4.5.1.1 Servlet 和 JSP 优化建议

1. 使用 init 方法和 ContextListener

Servlet 和 JSP 的 init 方法可用来缓存静态数据和资源引用。Web 容器在响应请求之前,会先初始化 Servlet。这个操作在 Servlet 的生命周期中只执行一次,所以 init()可以用来执行代价昂贵的一次性操作,包括静态内容的创建和缓存,比如在 J2EE1.4 的应用中,读取配置信息、初始化和缓存资源引用(包括 JNDI 查找 DataSource)。而在 JavaEE5 种,可以通过注入的方式访问资源,所以 init()不再需要用作此目的。

与 Servlet 的 init()类似,在 JSP 页面的初始化过程中,jspInit()方法只调用一次。在 JSP 中提供用户自定义的 jspInit(),可以生成一次性的操作。最常见的用途是创建和缓存静态数据。不过,jspInit()并不常用。

Servlet 的生命周期中会调用 ContextListener,可以用在应用程序特定数据的初始化和清扫阶段。

2. 使用恰当的 JSP include 机制

JSP 支持两种在页面中包含资源内容的方法。

include 指令方式。`<%@ include file="relativeURL" %>`将被包含文件的文本添加到页面中。这个包含过程是静态的，意味着文本会在 JSP 页面编译时合并进来。如果被包含的文件是 JSP 页面，它的 JSP 元素会被转换并包含到这个页面中。这种做法的缺点是，被包含文件的任何改动都不会反映到包含它的页面中，即便开启了动态重新加载 JSP 也是如此。只有当顶层页面发生变化重新生成被包含内容时，这些变化才可见。

include 标签方式。`<jsp:include page="relativeURL" />`支持为页面添加静态或动态的资源。如果是静态资源，它的内容（通过默认 Servlet 的调用获得）就会包含在调用页面中。如果是动态资源，调用的结果会包含在调用页面中。属性 `flush="true" | "false"` 可用来指定在包含资源调用页面的内容是否需要刷新。这个属性的默认值为 `false`。`<jsp:param>` 语句可以将一个或多个名/值对作为参数传递给被包含的资源。include 标签的动态性可以不依赖顶层页面的改动，就能使被包含页面的改动可见。

由于在编译时，include 声明包含引用资源的内容，这个机制可以改善包含 HTML 和其他静态内容的性能。另一方面，include 标签应该用于这样的场景，即需要包含的内容是引用资源的动态响应。

注意，如果是静态引用资源，我们可以使用 include 指令，而包含资源动态生成的响应，可以使用 include 标签。

3. 剔除空格

JSP 页面模板中的空格，无论有没有意义，都会被保留。这意味着一些无关紧要的字符，即便浏览器显示内容时不需要，也仍然会被 Web 容器处理并传送。

JSP 页面模板中保留的空格，会导致渲染输出中有成块的空格，降低 HTML 源文件的可读性。编码并传递这些无关字符还会产生性能开销。

剔除指令 `<%@ page trimDirectiveWhitespaces="true" %>` 可以消除多余的字符。声明需要加到所有需要剔除空格的页面中。另外，一组 JSP 的行为可以通过 `web.xml` 来配置。

在某些情况下，你不希望多余的字符增加传送成本，例如在低带宽网络上分发内容，服务器应该以最大压缩的形式生成内容，剥掉所有不必要的空格。一种选择是包含从输出中移除额外空格的 Servlet filter。添加 filter 会在请求处理的过程中增加额外处理成本。第二种选择是在部署前使用外部工具压缩 JSP。如果应用程序包括 CSS 和 JavaScript，那么 CSS 和 JavaScript 应该尽可能小，可以采取压缩的方式减少需要网络传递的整个文件大小。如果 Web 容器可以配置成发送压缩版 CSS 和 JavaScript，且浏览器支持压缩，那么开启压缩就能减少负荷了。

4. 使用 JSP: useBean

`jsp:useBean` 依据指定的名字和范围定位或初始化某个 Bean。这个标签支持两类初始化，`beanName` 和 `class`，还包括具有多种值得 `scope` 属性。重点是为这些属性选择适当的值，从而提供所需的功能和最佳的性能。

定位和初始化 Bean 所需要采用的步骤如下。

- (1) 尝试用你指定的范围和名称定为 bean。
- (2) 用你指定的名称定义一个对象引用变量。
- (3) 如果找到了 Bean，就将指向它的引用保存在上面的变量中。如果你指定了类型，则把该类型赋予那个 Bean。
- (4) 如果没有找到 Bean，依据你指定的类型初始化一个实例，将该实例的引用保存在新变量中。如果类名表示序列化模板，则用 `java.beans.Beans.instantiate` 初始化该 Bean。
- (5) 如果 `jsp:useBean` 已经初始化 Bean，并且它有 Body 标签或者元素（在 `<jsp:useBean>` 和 `</jsp:useBean>` 之间），则执行 body 标签。

Bean 的初始化取决于是否指定了 `class` 或 `beanName` 属性。如果指定 `class="package.class"`，则 Bean 用关键字 `new` 初始化。如果指定 `beanName="{package.class|<%=expression%}"`，则 Bean 从类、序列化模板、类或序列化模板的求值表达式初始化。用 `beanName` 可以为初始化需要的 Bean 提供灵活性（可以载运行时计算需要加载的类）。这种情况下，类由方法 `java.beans.Beans.instantiate` 初始化。如果 `beanName` 指定的值代表类或者序列化模板，则 Bean 的初始化包括 Classloader 加载资源。

`scope` 属性是指 Bean 存活的范围。它的值包括 `page`、`request`、`session` 及 `application`，默认为 `page`。你选择的 `scope` 值会影响性能。`application` 范围，Bean 只创建一次，初始化的代价分摊到应用的整个声明周期。但是，Bean 会增加应用程序的内存占用。

使用 `Session` 范围，只要会话是活跃的，Bean 会一直维护在内存中，也会增加内存占用。万一用户没有主动关闭会话，服务器就得在内存中一直维护会话直到会话超时。使用 `request` 或 `page` 范围时，页面调用时会创建新对象。在这些模式里，对象的存活期相对短，垃圾收集可以很快。然而，Bean 初始化的代价很高，会降低应用的整体性能。

5. 表达式语言

JSP2.0 支持表达式语言（EL），使得访问 JavaBean 的数据比较容易。Bean 可以用 `${name}` 语法访问，这个语法可以用在接受表达式的静态文本或任何自定义或标准的标签属性中。EL 可以用在 JSP 的 Scriptlet 或者 JSP 的表达式中。JSP EL 以及 JSP 标准标签库（JSTL）和自定义标签库使得开发复杂 JSP 要比用 Scriptlet 更容易。

从开发人员的角度来看，EL 是比 Scriptlet 更好的选择。但是，从性能角度看，EL 增加了解析变量名为对象和计算表达式的开销。而 Scriptlet 在编译阶段，就将必要的代码直接注入生成的 Servlet 中，不需要变量查找和复杂的表达式计算。由于引入 EL 带来的开销，EL JSP 页面的渲染时间比同样功能基于 Scriptlet 的页面略长。

性能上的差别取决于表达式的计算量和生成输出所涉及的其他工作量。例如，如果大部分时间花在列表生成上，渲染对象列表所带来的性能影响就会比较小。从另一方面来说，如果生成列表的代价不高，变量的计算代价就会凸显出来，从而导致 EL 的性能要比 Scriptlet 差。

6. HTTP 压缩

HTTP 压缩有助于减少文本数据从服务器传送到客户端的大小。如果浏览器支持压缩，服务器可以配置成传送压缩数据，然后在浏览器端解压。压缩量各有不同。Andy King 和 Konstantin

Balashov 已经表明 HTML 和 CSS 文件通常可以压缩到约 20%，JavaScript 文件平均压缩到 30%。

运送中压缩是低带宽与高 CPU、内存使用率之间的权衡，客户端和服务端都是如此。负荷越小，传送的成本越低，从而用户响应时间得以改善。对于通过慢速网络连接的客户端，传输延迟占据整个响应时间的比例很高，因而压缩可以改善这类客户端的性能。然后，压缩数据所消耗的 CPU 资源会降低 Web 容器的整体吞吐量。开启压缩时，客户端解压增加的成本也需要考虑。

4.5.1.2 内容缓存

现代 Web 应用的生成内容可以分为两大类：针对浏览用户的一般性页面和针对已知用户的定制页面。随着网站越来越受欢迎，支持的用户也达到几十万，这就需要为这些用户制定不同的缓存策略。一种常规的性能优化方式是缓存经常使用的内容，具体方式有如下几类。

1. 应用程序实现的动态页面缓存

应用程序把动态文件生成的 html 文件缓存到文件服务器，以后用户请求动态文件，直接从文件服务器加载对应的静态缓存的 html 文件返回给用户，这里面主要节省了动态语言的执行时间和数据库访问时间。但是增加了缓存框架的加载和缓存查找的时间。

2. 把解释执行的开发语言编译成为目标代码

这个主要把解释执行的高级语言，例如 java, php 直接编译成为平台相关的目标代码，汇编代码。在 java 里面，比较著名的就是即时编译器（JIT），其他的语言也要类似的机制。这里主要节省了解释执行代码的时间。但会增加即时编译的时间。

3. 利用反向代理服务器的缓存

利用类似 nginx 的反向代理服务器，对请求的 url 对应的输出进行缓存。这个缓存和应用程序实现的动态页面缓存类似，只不过用反向代理充当了应用程序的缓存实现。主要节省了动态缓存执行时间和数据库访问时间。

4. 客户端浏览器缓存

客户端浏览器缓存主要是通过通过在 http 头部增加 Last-Modified 等标识⁹，和服务器进行协商，是否是采用客户的本机缓存来实现。

4.5.2 死锁情况解决方案

Java 语言通过 synchronized 关键字来保证原子性，这是因为每一个 Object 都有一个隐含的锁，这个也称作监视器对象。在进入 synchronized 之前自动获取此内部锁，而一旦离开此方式，无论是完成或者中断都会自动释放锁。显然这是一个独占锁，每个锁请求之间是互斥的。相对于众多高级锁（Lock/ReadWriteLock 等），synchronized 的代价都比后者要高。但是 synchronized 的语法比较简单，而且也比较容易使用与理解。Lock 一旦调用了 lock() 方法获取到锁而未正确释放的话很有可能造成死锁，所以 Lock 的释放操作总是跟在 finally 代码块里面，这在代码结构上也是一次调整和冗余。Lock 的实现已经将硬件资源用到了极致，所以未来可优化的空间不大，除非硬件有了更高的性能，但是 synchronized 只是规范的一种实现，这在不同的平台不同的硬件还有很高的

⁹ 其它还包括 If-Modified-Since, Expires, Cache-Control 等。

提升空间, 未来 Java 锁上的优化也会主要在这上面。既然 `synchronized` 都不可能避免死锁¹⁰产生, 那么死锁情况会是经常容易出现的错误。

死锁问题是多线程特有的问题, 它可以被认为是线程间切换消耗系统性能的一种极端情况。在死锁时, 线程间相互等待资源, 而又不释放自身的资源, 导致无穷无尽的等待, 其结果是系统任务永远无法执行完成。死锁问题是在多线程开发中应该坚决避免和杜绝的问题。

一般来说, 要出现死锁问题需要满足以下条件。

- (1) 互斥条件: 一个资源每次只能被一个线程使用。
- (2) 请求与保持条件: 一个进程因请求资源而阻塞时, 对已获得的资源保持不放。
- (3) 不剥夺条件: 进程已获得的资源, 在未使用完之前, 不能强行剥夺。
- (4) 循环等待条件: 若干进程之间形成一种头尾相接的循环等待资源关系。

只要破坏死锁 4 个必要条件中的任何一个, 死锁问题就能被解决。

我们先来看一个示例, 前面说过, 死锁是两个甚至多个线程被永久阻塞时的一种运行局面, 这种局面的生成伴随着至少两个线程和两个或多个资源。代码清单 4-65 所示的示例中, 我们编写了一个简单的程序, 它将会引起死锁发生, 然后我们就会明白如何分析它。

代码清单 4-65 死锁示例 1

```
public class ThreadDeadlock {
    public static void main(String[] args) throws InterruptedException {
        Object obj1 = new Object();
        Object obj2 = new Object();
        Object obj3 = new Object();
        Thread t1 = new Thread(new SyncThread(obj1, obj2), "t1");
        Thread t2 = new Thread(new SyncThread(obj2, obj3), "t2");
        Thread t3 = new Thread(new SyncThread(obj3, obj1), "t3");
        t1.start();
        Thread.sleep(5000);
        t2.start();
        Thread.sleep(5000);
        t3.start();
    }
}

class SyncThread implements Runnable{
    private Object obj1;
    private Object obj2;
    public SyncThread(Object o1, Object o2){
        this.obj1=o1;
        this.obj2=o2;
    }
}
```

¹⁰ 死锁是操作系统层面的一个错误, 是进程死锁的简称, 最早在 1965 年由 Dijkstra 在研究银行家算法时提出的, 它是计算机操作系统乃至整个并发程序设计领域最难处理的问题之一。


```

@Override
public void run() {
    String name = Thread.currentThread().getName();
    System.out.println(name + " acquiring lock on "+obj1);
    synchronized (obj1) {
        System.out.println(name + " acquired lock on "+obj1);
        work();
        System.out.println(name + " acquiring lock on "+obj2);
        synchronized (obj2) {
            System.out.println(name + " acquired lock on "+obj2);
            work();
        }
        System.out.println(name + " released lock on "+obj2);
    }
    System.out.println(name + " released lock on "+obj1);
    System.out.println(name + " finished execution.");
}
private void work() {
    try {
        Thread.sleep(30000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

在上面的程序中同步线程正完成 `Runnable` 的接口，它工作的是两个对象，这两个对象向对方寻求死锁而且都在使用同步阻塞。在主函数中，我使用了三个为同步线程运行的线程，而且在其中每个线程中都有一个可共享的资源。这些线程以向第一个对象获取封锁这种方式运行。但是当它试着向第二个对象获取封锁时，它就会进入等待状态，因为它已经被另一个线程封锁住了。这样，在线程引起死锁的过程中，就形成了一个依赖于资源的循环。当我执行上面的程序时，就产生了输出，但是程序却因为死锁无法停止。

JVM 提供了一些工具可以来帮助诊断死锁的发生，如下面程序清单 4-66 所示，我们实现了一个死锁，然后尝试通过 `jstack` 命令追踪、分析死锁发生。

代码清单 4-66 死锁代码 2

```

import java.util.concurrent.locks.ReentrantLock;
//下面演示一个简单的死锁，两个线程分别占用 south 锁和 north 锁，并同时请求对方占用的锁，导致死锁
public class DeadLock extends Thread{
    protected Object myDirect;
    static ReentrantLock south = new ReentrantLock();
    static ReentrantLock north = new ReentrantLock();
    public DeadLock(Object obj){
        this.myDirect = obj;
        if(myDirect==south){

```

```
this.setName("south");
}else{
this.setName("north");
}
}
@Override
public void run(){
if(myDirect==south){
try{
north.lockInterruptibly();//占用 north
try{
Thread.sleep(500);
}catch(Exception ex){
ex.printStackTrace();
}
south.lockInterruptibly();
System.out.println("car to south has passed");
}catch(InterruptedException ex){
System.out.println("car to south is killed");
ex.printStackTrace();
}finally{
if(north.isHeldByCurrentThread()){
north.unlock();
}
if(south.isHeldByCurrentThread()){
south.unlock();
}
}
}
if(myDirect==north){
try{
south.lockInterruptibly();//占用 south
try{
Thread.sleep(500);
}catch(Exception ex){
ex.printStackTrace();
}
north.lockInterruptibly();
System.out.println("car to north has passed");
}catch(InterruptedException ex){
System.out.println("car to north is killed");
ex.printStackTrace();
}finally{
if(north.isHeldByCurrentThread()){
north.unlock();
}
if(south.isHeldByCurrentThread()){
south.unlock();
}
```



```

    }
    }
    }

    }

    public static void main(String[] args) throws InterruptedException{
        DeadLock car2south = new DeadLock(south);
        DeadLock car2north = new DeadLock(north);
        car2south.start();
        car2north.start();
    }
}

```

jstack 可用于导出 Java 应用程序的线程堆栈，-l 选项用于打印锁的附加信息。我们运行 jstack 命令，输出清单 4-67 和 4-68 所示，其中代码清单 4-67 里面可以看到线程处于运行状态，代码中调用了拥有锁投票、定时锁等候和可中断锁等候等特性的 ReentrantLock 锁机制。

代码清单 4-67 jstack 运行输出

```

[root@facenode4 ~]# jstack -l 31274
2015-01-29 12:40:27
Full thread dump Java HotSpot(TM) 64-Bit Server VM (20.45-b01 mixed mode):
"Attach Listener" daemon prio=10 tid=0x00007f6d3c001000 nid=
    0x7a87 waiting on condition [0x0000000000000000]
java.lang.Thread.State: RUNNABLE
Locked ownable synchronizers:
- None
"DestroyJavaVM" prio=10 tid=0x00007f6da4006800 nid=
    0x7a2b waiting on condition [0x0000000000000000]
java.lang.Thread.State: RUNNABLE
Locked ownable synchronizers:
- None
"north" prio=10 tid=0x00007f6da4101800 nid=
    0x7a47 waiting on condition [0x00007f6d8963b000]
java.lang.Thread.State: WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <0x0000000075903c7c8> (
a java.util.concurrent.locks.ReentrantLock$NonfairSync)
at java.util.concurrent.locks.LockSupport.park(LockSupport.java:156)
at java.util.concurrent.locks.AbstractQueuedSynchronizer.
parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:811)
at java.util.concurrent.locks.AbstractQueuedSynchronizer.
doAcquireInterruptibly(AbstractQueuedSynchronizer.java:867)
at java.util.concurrent.locks.AbstractQueuedSynchronizer.
acquireInterruptibly(AbstractQueuedSynchronizer.java:1201)
at java.util.concurrent.locks.ReentrantLock.lockInterruptibly(ReentrantLock.
java:312)
at DeadLock.run(DeadLock.java:50)
Locked ownable synchronizers:

```

```

- <0x000000075903c798> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)
"south" prio=10 tid=0x00007f6da4100000 nid=
                                0x7a46 waiting on condition [0x00007f6d8973c000]
java.lang.Thread.State: WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <0x000000075903c798> (
a java.util.concurrent.locks.ReentrantLock$NonfairSync)
at java.util.concurrent.locks.LockSupport.park(LockSupport.java:156)
at java.util.concurrent.locks.AbstractQueuedSynchronizer.
parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:811)
at java.util.concurrent.locks.AbstractQueuedSynchronizer.
doAcquireInterruptibly(AbstractQueuedSynchronizer.java:867)
at java.util.concurrent.locks.AbstractQueuedSynchronizer.
acquireInterruptibly(AbstractQueuedSynchronizer.java:1201)
at java.util.concurrent.locks.ReentrantLock.lockInterruptibly(ReentrantLock.
java:312)
at DeadLock.run(DeadLock.java:28)
Locked ownable synchronizers:
- <0x000000075903c7c8> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)

"Low Memory Detector" daemon prio=10 tid=0x00007f6da40d2800 nid=
0x7a44 runnable [0x0000000000000000]
java.lang.Thread.State: RUNNABLE
Locked ownable synchronizers:
- None

"C2 CompilerThread1" daemon prio=10 tid=0x00007f6da40d0000 nid=
0x7a43 waiting on condition [0x0000000000000000]
java.lang.Thread.State: RUNNABLE
Locked ownable synchronizers:
- None

"C2 CompilerThread0" daemon prio=10 tid=0x00007f6da40cd000 nid=
0x7a42 waiting on condition [0x0000000000000000]
java.lang.Thread.State: RUNNABLE
Locked ownable synchronizers:
- None

"Signal Dispatcher" daemon prio=10 tid=0x00007f6da40cb000 nid=
0x7a41 runnable [0x0000000000000000]
java.lang.Thread.State: RUNNABLE
Locked ownable synchronizers:
- None

"Finalizer" daemon prio=10 tid=0x00007f6da40af000 nid=
0x7a40 in Object.wait() [0x00007f6d89d44000]
java.lang.Thread.State: WAITING (on object monitor)
at java.lang.Object.wait(Native Method)
- waiting on <0x0000000759001300> (a java.lang.ref.ReferenceQueue$Lock)
at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:118)
- locked <0x0000000759001300> (a java.lang.ref.ReferenceQueue$Lock)
at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:134)

```



```

at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:171)
Locked ownable synchronizers:
- None
"Reference Handler" daemon prio=10 tid=0x00007f6da40ad000 nid=
0x7a3f in Object.wait() [0x00007f6d89e45000]
java.lang.Thread.State: WAITING (on object monitor)
at java.lang.Object.wait(Native Method)
- waiting on <0x00000007590011d8> (a java.lang.ref.Reference$Lock)
at java.lang.Object.wait(Object.java:485)
at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:116)
- locked <0x00000007590011d8> (a java.lang.ref.Reference$Lock)
Locked ownable synchronizers:
- None

"VM Thread" prio=10 tid=0x00007f6da40a6000 nid=0x7a3e runnable
"GC task thread#0 (ParallelGC)" prio=10 tid=0x00007f6da4019800 nid=0x7a2c runnable
"GC task thread#1 (ParallelGC)" prio=10 tid=0x00007f6da401b000 nid=0x7a2d runnable
"GC task thread#2 (ParallelGC)" prio=10 tid=0x00007f6da401d000 nid=0x7a2e runnable
"GC task thread#3 (ParallelGC)" prio=10 tid=0x00007f6da401f000 nid=0x7a2f runnable
"GC task thread#4 (ParallelGC)" prio=10 tid=0x00007f6da4020800 nid=0x7a30 runnable
"GC task thread#5 (ParallelGC)" prio=10 tid=0x00007f6da4022800 nid=0x7a31 runnable
"GC task thread#6 (ParallelGC)" prio=10 tid=0x00007f6da4024800 nid=0x7a32 runnable
"GC task thread#7 (ParallelGC)" prio=10 tid=0x00007f6da4026000 nid=0x7a33 runnable
"GC task thread#8 (ParallelGC)" prio=10 tid=0x00007f6da4028000 nid=0x7a34 runnable
"GC task thread#9 (ParallelGC)" prio=10 tid=0x00007f6da402a000 nid=0x7a35 runnable
"GC task thread#10 (ParallelGC)" prio=10 tid=0x00007f6da402b800 nid=0x7a36 runnable
"GC task thread#11 (ParallelGC)" prio=10 tid=0x00007f6da402d800 nid=0x7a37 runnable
"GC task thread#12 (ParallelGC)" prio=10 tid=0x00007f6da402f800 nid=0x7a38 runnable
"GC task thread#13 (ParallelGC)" prio=10 tid=0x00007f6da4031000 nid=0x7a39 runnable
"GC task thread#14 (ParallelGC)" prio=10 tid=0x00007f6da4033000 nid=0x7a3a runnable
"GC task thread#15 (ParallelGC)" prio=10 tid=0x00007f6da4035000 nid=0x7a3b runnable
"GC task thread#16 (ParallelGC)" prio=10 tid=0x00007f6da4036800 nid=0x7a3c runnable
"GC task thread#17 (ParallelGC)" prio=10 tid=0x00007f6da4038800 nid=0x7a3d runnable

"VM Periodic Task Thread" prio=10 tid=0x00007f6da40dd000 nid=0x7a45 waiting on
condition
JNI global references: 886

```

代码清单 4-68 直接打印出出现死锁的情况，报告 north 和 south 两个线程互相等待资源，出现了死锁。

代码清单 4-68 jstack 运行输出片段

```

Found one Java-level deadlock:
=====
"north":
  waiting for ownable synchronizer 0x000000075903c7c8, (
  a java.util.concurrent.locks.ReentrantLock$NonfairSync),
  which is held by "south"

```

```

"south":
  waiting for ownable synchronizer 0x000000075903c798, (
  a java.util.concurrent.locks.ReentrantLock$NonfairSync,
  which is held by "north"
Java stack information for the threads listed above:
=====
"north":
  at sun.misc.Unsafe.park(Native Method)
  - parking to wait for <0x000000075903c7c8> (
  a java.util.concurrent.locks.ReentrantLock$NonfairSync)
  at java.util.concurrent.locks.LockSupport.park(LockSupport.java:156)
  at java.util.concurrent.locks.AbstractQueuedSynchronizer.
  parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:811)
  at java.util.concurrent.locks.AbstractQueuedSynchronizer.
  doAcquireInterruptibly(AbstractQueuedSynchronizer.java:867)
  at java.util.concurrent.locks.AbstractQueuedSynchronizer.
  acquireInterruptibly(AbstractQueuedSynchronizer.java:1201)
  at java.util.concurrent.locks.ReentrantLock.lockInterruptibly(ReentrantLock.
  java:312)
  at DeadLock.run(DeadLock.java:50)
"south":
  at sun.misc.Unsafe.park(Native Method)
  - parking to wait for <0x000000075903c798> (
  a java.util.concurrent.locks.ReentrantLock$NonfairSync)
  at java.util.concurrent.locks.LockSupport.park(LockSupport.java:156)
  at java.util.concurrent.locks.AbstractQueuedSynchronizer.
  parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:811)
  at java.util.concurrent.locks.AbstractQueuedSynchronizer.
  doAcquireInterruptibly(AbstractQueuedSynchronizer.java:867)
  at java.util.concurrent.locks.AbstractQueuedSynchronizer.
  acquireInterruptibly(AbstractQueuedSynchronizer.java:1201)
  at java.util.concurrent.locks.ReentrantLock.lockInterruptibly(ReentrantLock.
  java:312)
  at DeadLock.run(DeadLock.java:28)
Found 1 deadlock.

```

死锁是由四个必要条件导致的，所以一般来说，只要破坏这四个必要条件中的一个条件，死锁情况就不会发生。

- (1) 如果想要打破互斥条件，我们需要允许进程同时访问某些资源，这种方法受制于实际场景，不太容易实现条件。
- (2) 打破不可抢占条件，这样需要允许进程强行从占有者那里夺取某些资源，或者简单一点理解，占有资源的进程不能再申请占有其他资源，必须释放手上的资源之后才能发起申请，这个其实也很难找到适用场景。
- (3) 进程在运行前申请得到所有的资源，否则该进程不能进入准备执行状态。这个方法看似有点用处，但是它的缺点是可能导致资源利用率和进程并发性降低。

(4) 避免出现资源申请环路,即对资源事先分类编号,按号分配。这种方式可以有效提高资源的利用率和系统吞吐量,但同时增加了系统开销,增大了进程对资源的占用时间。

如果我们在死锁检查时发现了死锁情况,那么就要努力消除死锁,使系统从死锁状态中恢复过来。消除死锁的几种方式如下。

(1) 最简单、最常用的方法就是进行系统的重新启动,不过这种方法代价很大,它意味着在这之前所有的进程已经完成的计算工作都将付之东流,包括参与死锁的那些进程,以及未参与死锁的进程。

(2) 撤销进程,剥夺资源。终止参与死锁的进程,收回它们占有的资源,从而解除死锁。这时又分两种情况:一次性撤销参与死锁的全部进程,剥夺全部资源;或者逐步撤销参与死锁的进程,逐步收回死锁进程占有的资源。一般来说,选择逐步撤销的进程时要按照一定的原则进行,目的是撤销那些代价最小的进程,比如按进程的优先级确定进程的代价;考虑进程运行时的代价和与此进程相关的外部作业的代价等因素。

(3) 进程回退策略,即让参与死锁的进程回退到没有发生死锁前某一点处,并由此点处继续执行,以求再次执行时不再发生死锁。虽然这是个较理想的办法,但是操作起来系统开销极大,要有堆栈这样的机构记录进程的每一步变化,以便今后的回退,有时无法做到这些。

MySQL 死锁情况解决方法

假设我们用 Show innodb status 检查引擎状态时发现了死锁情况,如代码清单 4-69 所示。

代码清单 4-69 MySQL 死锁

```
WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 843102 n bits 600 index `KEY_TSKTASK_MONTIME2` of table
`dcnet_db/TSK_TASK` trx id 0 677833454 lock_mode X locks rec but not gap waiting
Record lock, heap no 395 PHYSICAL RECORD: n_fields 3; compact format; info bits 0
0: len 8; hex 8000000000000425; asc %;; 1: len 8; hex 800012412c66d29c;
asc A,f ;; 2: len 8; hex 800000000097629c; asc b ;;
*** WE ROLL BACK TRANSACTION
```

我们假设相关的数据表上面有一个索引,这次的死锁就是由于两条记录同时访问到了相同的索引造成的。

我们首先来看看 InnoDB 类型的数据表,只要能够解决索引问题,就可以解决死锁问题。MySQL 的 InnoDB 引擎是行级锁,需要注意的是,这不是对记录进行锁定,而是对索引进行锁定。在 UPDATE、DELETE 操作时,MySQL 不仅锁定 WHERE 条件扫描过的所有索引记录,而且会锁定相邻的键值,即所谓的 next-key locking。如语句 UPDATE TSK_TASK SET UPDATE_TIME = NOW() WHERE ID > 10000 会锁定所有主键大于等于 1000 的所有记录,在该语句完成之前,你就不能对主键等于 10000 的记录进行操作;当非簇索引(non-cluster index)记录被锁定时,相关的簇索引(cluster index)记录也需要被锁定才能完成相应的操作。

再分析一下发生问题的两条 SQL 语句,当“update TSK_TASK set STATUS_ID=1064,UPDATE_TIME=now() where STATUS_ID=1061 and MON_TIME<date_sub(now(), INTERVAL 30 minute)”执

行时，MySQL 会使用 KEY_TSKTASK_MONTIME2 索引，因此首先锁定相关的索引记录，因为 KEY_TSKTASK_MONTIME2 是非簇索引，为执行该语句，MySQL 还会锁定簇索引（主键索引）。

假设“update TSK_TASK set STATUS_ID=1067,UPDATE_TIME=now() where ID in (9921180)”几乎同时执行时，本语句首先锁定簇索引(主键)，由于需要更新 STATUS_ID 的值，所以还需要锁定 KEY_TSKTASK_MONTIME2 的某些索引记录。

这样第一条语句锁定了 KEY_TSKTASK_MONTIME2 的记录，等待主键索引，而第二条语句则锁定了主键索引记录，而等待 KEY_TSKTASK_MONTIME2 的记录，这样死锁就产生了。

我们通过拆分第一条语句解决了死锁问题：即先查出符合条件的 ID：select ID from TSK_TASK where STATUS_ID=1061 and MON_TIME < date_sub(now(), INTERVAL 30 minute); 然后再更新状态：update TSK_TASK set STATUS_ID=1064 where ID in (...)。

4.5.3 JavaBeans 组件

JavaBeans 是 Java 平台上的组件模型。要在 Java 平台上创建可复用的组件，应该遵循 JavaBeans 的规范。对于 JavaBeans 组件，开发人员比较熟悉的是 Java EE 中的 EJB（Enterprise JavaBeans），以及 Java 类中遵循 JavaBeans 命名规范的属性获取和设置的方法。JavaBeans 的强大之处在于以规范的组件模型作为基础，可以通过工具很方便地进行单个组件的自定义和多个组件的组装等操作。对于所有遵循 JavaBeans 规范的组件，都可以通过工具以统一的方式来进行操作。

符合 JavaBeans 规范的每个组件都包含 3 类信息，分别是属性、方法和事件。属性指的是一个组件暴露出来的外观或行为上的特征。可以通过改变数据的值来定制组件的外观或行为。JavaBeans 组件的方法与一般的 Java 方法并没有区别，可以在其他组件中调用。事件是组件之间进行交互的方式。某个组件可以发布事件，而另外的组件可以在这个事件上注册监听器。

一个 JavaBeans 组件可以通过 java.beans.Introspector 类来获取组件中的属性、方法和事件的信息，使用的是 Introspector 类中的静态方法 getBeanInfo。该方法的返回值是包含了组件相关信息的 java.beans.BeanInfo 接口的实现对象。获取组件信息的方式可以有两种，一种是开发人员自己提供的 BeanInfo 接口的实现类，另外一种是由系统通过反射 API 来自动发现组件中的信息。对于第一种方式，系统会根据固定的名称模式来查找组件对应的 BeanInfo 接口的实现类。比如，对于类名为“com.java7book.My”的组件，查找类名为“com.java7book.MyBeanInfo”的 BeanInfo 接口的实现类作为组件的信息来源。如果没有找到相关实现类，会使用第二种方式，即通过反射 API 来发现相关信息。这两种方式的一个重要区别在于，在找到 BeanInfo 接口的实现类之后，不会再继续查找该组件的父类来获取信息；而通过反射 API 的方式则会沿着继承层次结构树一直向上查找父类中的相关信息。

对于具体的组件信息的获取过程，getBeanInfo 方法也提供了不同的重载方式供开发人员进行配置。可以配置的内容主要有两个，一个是在获取过程中包含哪些类中的信息。前面提到过，组件的父类中的信息有可能被包含进来，如果不希望包含组件的某些父类中的信息，那么可以指明终止组件信息获取过程的类名。当沿着继承层次结构树向上获取时，如果遇到指定的终止类，就停止继续获取。另外一个可配置的内容是对找到的 BeanInfo 接口实现类的处理方式。在 getBeanInfo 方法中允许设置 3 种不同的处理方式，对应的参数值分别是使用所有 BeanInfo 接口实现类的 USE_ALL_BEANINFO、忽略组件类对应的 BeanInfo 接口实现类的 IGNORE_IMMEDIATE_

BEANINFO 和忽略包括组件类的父类在内的所有 BeanInfo 接口实现类的 IGNORE_ALL_BEANINFO。

通过这两种配置方式，可以很好地控制组件信息的获取过程。不过在 Java7 之前，这两种配置方式不能同时使用，只能使用其中一种。Java7 添加了额外的 getBeanInfo 的重载方式。

JavaBeans 组件也提供了动态执行语句和表达式的能力，主要是为了方便工具的使用者以类似脚本语言的方式来对组件进行操作，比如调用一个组件对象 myBean 的 open 方法的语句可以直接写成“myBean.open()”。语句和表达式的区别在于，语句不关心具体的执行结果，而表达式则会把执行结果记录下来。语句和表达式分别用 java.beans.Statement 和 java.beans.Expression 类来表示。Expression 类继承自 Statement 类，并添加了获取和设置执行结果的方法。Statement 类的 execute 方法用来执行语句。在 Java7 中，Expression 类增加了缓存执行结果的功能。当通过 Expression 类的 getValue 来获取执行结果时，如果 execute 方法之前没有被调用过，则会先调用 execute 方法，再返回结果，同时也会把执行结果记录下来。

JavaBeans 组件在其属性发生变化之后，可以被持久化，以保存组件的内部状态。之后再次需要时可以把保存的内容再次读取处理，并恢复组件的内部状态。JavaBeans 组件的持久化依赖的是 Java 标准的对象序列化机制。一般来说，可以把组件的内部状态以流的二进制形式保存，或者保存成 XML 文件。以流的形式进行持久化时使用的是 Java 中的 java.io.ObjectInputStream 和 java.io.ObjectOutputStream 类，而在以 XML 文件作为持久化形式时，使用的是 java.beans.XMLEncoder 和 java.beans.XMLDecoder 类。

Java7 对将 JavaBeans 组件保存成 XML 文档的功能做了更新，在 XMLEncoder 类中增加了构造方法，可以更加精细地控制保存时的行为。Java7 之前的 XMLEncoder 构造方法只接受一个 java.io.OutputStream 类的对象作为参数，表示保存内容的输出流。而 Java7 新增的构造方法中添加了额外的 3 个参数，分别是输出时使用的字符集、是否是输出 XML 处理指令声明和整个 XML 文档的缩进空格数。允许指定输出时使用的字符集主要是为了满足不同的编码格式需求，而另外两个参数是为了使输出的 XML 文档内容可以被嵌入到其他 XML 文档中。

用于读取 XMLEncoder 类的输出文档的 XMLDecoder 类也有一些更新，主要是提供了更好的对 SAX 解析方式的支持。XMLDecoder 类新增了一个接受 org.xml.sax.InputSource 类型参数的构造方法。在 Java7 之前，创建 XMLDecoder 类的对象时，只能使用 InputStream 类的对象来表示解析时的输入数据。而 InputSource 类则提供了更加丰富的方式来表示输入数据，除了 InputStream 类的对象之外，还支持使用标示符和 java.io.Reader 类的对象。

4.6 本章小结

本章首先针对算法相关的概念、优化建议进行了陈述，然后挑选了一些较有代表性的设计模式进行了深入的优化建议介绍，接下来对网络相关、数据库相关的优化建议也做了一些技术、经验分享，最后对程序设计过程中遇到的示例、常见问题做了一些总结和分享。

第 5 章 Java 并程序优化建议

“优先发展公共交通，加强财政保障，确立公共交通引领和支撑城市交通发展的格局；加快推进高架、主干道、快速路、停车场（库）及地铁、轻轨等城市交通基础设施建设，大力推进停车产业化发展，构建立体交通；强化城市交通管理，提高城市交通运行效能。”这段话出自《中国交通报》发表的文章“浙江综合治堵提升城市生活品质”。

程序设计优化与交通治理拥堵有一定的相似性，都可以开拓程序运行线路，多通道、多维度同时运行程序，这些并行方法可以帮助提高程序运行速度，本章我们就来针对 Java 并程序优化建议讨论。

本章主要介绍和解决以下问题，并行计算是 Java 程序员较难掌握的技术：

- 什么是多线程编程及优化方式。
- 如何增加程序并行性。
- 如何调优锁设计机制。
- JDK 类库里面提供了哪些有用的方式可以加强并发。

5.1 并程序优化概述

我们知道，即使是单核处理器也支持多线程执行代码，CPU 通过给每个线程分配 CPU 时间片的方式来实现这个机制。时间片方式是 CPU 分配给各个线程的时间长度，因为每一片时间片非常短，一般是几十毫秒，所以 CPU 通过不停地切换线程执行，让我们感觉多个线程是同时在执行的，事实上它们是不停地被轮转切换着运行的，这在第 8 章关于 CGroup 技术的实践中可以看到，分配到线程的 CPU 核是不停地变换的，并不是锁定在某一个核上。

既然 CPU 通过时间片分配算法来循环执行任务，那么当前任务执行一个时间片后切换到下一个任务。但是在切换前后保存上一个任务的状态，以便下次切换回这个任务时，可以再加载这个

任务的状态。所以任务从保存到再加载的过程就是一次上下文切换。这就好比我们同时读一本英文原版书，当发现不认识的单词时，我们需要查询字典，字典查询完毕后我们还是可以回到英文书继续读下去，整个过程就是所谓的切换过程。

注意，由于并行程序与串行程序的不同特点，又由于适用于串行程序的一些数据结构不是线程安全的，所以它们并不能直接用于并行环境下正常工作，对于什么是线程不安全，我们会在 5.1.4 节举例说明。

5.1.1 资源限制带来的挑战

资源限制是指在进行并发编程时，程序的执行速度受限于计算机硬件资源或软件资源。例如，服务器的带宽只有 2Mb/s，某个资源的下载速度是 1Mb/s 每秒，系统启动 10 个线程下载资源，下载速度不会变成 10Mb/s，所以在进行并发编程时，要考虑这些资源的限制。硬件资源限制主要有带宽的上传/下载速度、硬盘读写速度和 CPU 的处理速度。软件资源限制主要有数据库的连接数和 Socket 连接数等。

在并发编程中，将代码执行速度加快的原则和做法通常是将代码中串行执行的部分变成并行执行。但是如果将某段串行的代码并行执行，因为资源受限，所以仍然可能是串行执行，这时候程序不仅不会加快执行，反而会更慢，因为程序的执行过程中增加了上下文切换和资源调度的时间。例如，一段程序使用多线程在办公网内部并发下载和处理数据，导致 CPU 利用率达到 100%，几个小时都不能完成任务，改成单线程后一个小时就可执行完成。

对于硬件资源限制，可以考虑使用集群并行执行程序。通俗点来说就是，既然单机的资源有限制，那么就让程序在多台机器上运行，比如使用 Mesos 框架¹搭建服务器集群，不同的机器处理不同的数据。

同集中式系统相比较，分布式系统的一个潜在的优势在于它的高可靠性。通过把工作负载分散到众多的机器上，单个芯片故障最多只会使一台机器停机，其他机器不会受任何影响。理想条件下，某一时刻如果有 5% 的计算机出现故障，系统仍能继续工作，只是损失 5% 的性能。对于关键性的应用，如核反应堆或飞机的控制系统，采用分布式系统来实现主要是考虑到它可以获得高可靠性。

在现代集群里，不同的框架所要求的计算需求非常不同，企业需要运行多种框架，并在其间共享数据和资源。资源管理程序面临巨大的挑战和互为矛盾的目标。

- **高效性：**高效共享资源是集群管理软件的终极目标。
- **隔离性：**当多个任务共享资源时，最重要的考量之一是确保资源的隔离性。隔离性和正确调度的整合是保证服务级别协议（SLA）的基础。
- **可伸缩性：**现代基础架构的持续增长要求集群管理程序可以线性伸缩。一个重要的可伸缩性指标是框架伸缩决策制定所需的延时。
- **健壮性：**集群管理是中央组件，持续的业务运营要求健壮的集群管理。从良好测试的代码

¹ Mesos 是集群管理器，力争通过在多种框架之间动态共享资源来优化资源使用率。该项目于 2009 年由位于 Berkeley 的加利福尼亚大学发起，已经在很多公司的生产环境上使用过，包括 Twitter 和 Airbnb。2013 年 7 月，在该项目孵化大概两年时，就成为 Apache 的最高级别项目。

到容错设计，很多方面都有助于提高健壮性。

- **可扩展性**：在任何公司里，集群管理软件的开发量都非常大，而且这些软件可能已经使用了数十年。运营期间，企业政策和/或硬件的变化都不可避免地要求集群资源管理方式的改变，因此，对于大型企业来说，可维护性非常重要。集群管理软件必须是可配置的，同时考虑到约束条件（比如位置、硬件等），并且需要能够支持多种框架。

对于软件资源限制，可以考虑使用资源池将资源复用，以此提高资源的复用率。从较小的概念来理解，比如使用连接池将数据库和 Socket 连接复用，或者在调用对方 WebService 接口获取数据时，同时只建立一个连接。从较大的概念来讲，将计算任务分布在可动态升级和被虚拟化的资源池上，用户可以通过网络很方便地按需获取计算力、存储空间和信息服务。这种新的基于网络的运算方式，通过网络为用户提供可按需使用的服务。它通过分布式处理、虚拟化、在线软件等技术将数据中心的计算、存储、网络等 IT 基础设施，以及开发平台、软件等服务抽象成可运营、可管理的 IT 资源，通过互联网动态提供给用户，用户按实际使用数量进行付费，这就是云计算的理念。

第 2 章里面曾经介绍过，传统的冯·诺依曼型结构已经无法满足要求，计算机网络的出现使分布式系统成为可能，并得到飞速发展和应用。分布式系统是并行计算的有力推动，所谓分布式系统是指由多个相互连接的处理资源组成的计算机系统，计算机之间通过消息传递进行通信和动作协调，从用户角度来看，它如同一个集中的单机系统。大多数分布式系统建立在计算机网络之上，网络中的计算机可能在空间上有距离，可能在不同的国家和地区，也可能在同一栋楼房或同一个房间。分布式系统中的每个节点既独立工作，又与其他所有节点并行工作。每个节点多于一个进程（执行程序），每个进程多于一个线程（并行执行任务），可在系统中充当组件。大多数组件具有反应性，对来自用户的命令和来自其他组件的消息不断地进行响应。就像我们每天都要使用的操作系统一样，分布式系统旨在避免服务终止，因此应该始终保持至少部分可用的状态。

分布式系统中服务和应用两者均可提供被客户共享的资源。因此，可能几个客户同时试图访问同一个共享的资源。例如，学校选课系统的数据库在选课时可能被非常频繁地访问。最简单的办法是管理共享资源的进程在一个时刻接受一个客户请求。但这种方法限制了吞吐量，因此服务和应用通常允许并发地处理多个客户的请求。为了详细说明这个问题，我们假设每个资源被封装成一个对象，调用在并发线程中执行。在这种情况下，几个线程可能在一个对象内并发地执行，它们对于对象的操作可能相互冲突，产生不一致的结果。为了使对象在并发环境中能安全使用，对象的同步操作必须保持数据的一致性。这可以通过标准的技术，如在大多数操作系统使用的信号量技术来实现。

5.1.2 进程、线程、协程

进程是操作系统结构的基础，是一次程序的执行，是一个程序及其数据在处理机上顺序执行时所发生的活动，是程序在一个数据集合上运行的过程。总而言之，进程是系统进行资源分配和调度的一个独立单位。

在 Java 程序中可能需要调用底层操作系统上的其他程序，Java 标准 API 提供了创建底层操作系统上运行的进程的能力，只需要传入正确的命令和相关的参数，就可以启动一个进程。在进程启动之后，可以从 Java 程序对进程提供输入数据，以及读取进程运行过程中产生的输出数据。对

于在 Java 程序中启动其他进程这个任务来说,最重要的是输入和输出的处理。通常的做法是把 Java 程序的内部运行结果作为输入传递给一个新创建的进程,然后等待进程执行完成。在得到进程输出的运行结果之后,再继续下面的处理。通过这种方式,底层操作系统上的其他进程可以很好地与 Java 程序集成起来。

在 Java7 之前,对进程的输入和输出进行处理的方式比较有限,只支持管道式的方式。进程的输入对 Java 程序来说是一个输出流,程序向这个输出流中写入的数据会通过管道传递给进程。同样的,进程的输出对于 Java 程序来说是一个输出流,通过读取此输入流的内容获得进程的输出。标准的创建新进程的过程是使用 `java.lang.ProcessBuilder` 类来设置新进程的属性,然后通过 `start` 方法来启动进程的执行。`ProcessBuilder` 类的 `start` 方法的返回值是一个表示进程的 `java.lang.Process` 类的对象。通过 `Process` 类的 `getOutputStream` 方法可以得到向进程写入数据的输出流,而通过 `getInputStream` 和 `getErrorStream` 方法可以分别得到包含进程正常执行和出错时输出内容的输入流。下面示例代码启动了 Windows 上的命令行工具来执行“`netstat -a`”命令,并把结果保存到一个文件中。

代码清单 5-1 启动命令行工具

```
import java.io.IOException;
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;

public class StartProcess {
    public void startProcessNormal() throws IOException{
        ProcessBuilder pb = new ProcessBuilder("cmd.exe", "/c", "netstat", "-a");
        Process process = pb.start();
        InputStream inputstream = process.getInputStream();
        Files.copy(inputstream, Paths.get("netstat.txt"), StandardCopyOption.
REPLACE_EXISTING);
    }

    public static void main(String[] args){
        StartProcess sp = new StartProcess();
        try {
            sp.startProcessNormal();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

程序运行输出如清单 5-2 所示。

代码清单 5-2 清单 5-1 运行输出

活动连接

协议	本地地址	外部地址	状态
TCP	0.0.0.0:135	NB-ZHOUMINGYAO:0	LISTENING
TCP	0.0.0.0:443	NB-ZHOUMINGYAO:0	LISTENING
TCP	0.0.0.0:445	NB-ZHOUMINGYAO:0	LISTENING
TCP	0.0.0.0:902	NB-ZHOUMINGYAO:0	LISTENING
TCP	0.0.0.0:912	NB-ZHOUMINGYAO:0	LISTENING
TCP	0.0.0.0:1025	NB-ZHOUMINGYAO:0	LISTENING
TCP	0.0.0.0:1026	NB-ZHOUMINGYAO:0	LISTENING
TCP	0.0.0.0:1027	NB-ZHOUMINGYAO:0	LISTENING
TCP	0.0.0.0:1028	NB-ZHOUMINGYAO:0	LISTENING
TCP	0.0.0.0:1031	NB-ZHOUMINGYAO:0	LISTENING
TCP	0.0.0.0:1032	NB-ZHOUMINGYAO:0	LISTENING
TCP	0.0.0.0:5904	NB-ZHOUMINGYAO:0	LISTENING
TCP	0.0.0.0:7909	NB-ZHOUMINGYAO:0	LISTENING
TCP	0.0.0.0:17866	NB-ZHOUMINGYAO:0	LISTENING
TCP	127.0.0.1:8307	NB-ZHOUMINGYAO:0	LISTENING
TCP	127.0.0.1:10000	NB-ZHOUMINGYAO:0	LISTENING
TCP	127.0.0.1:27018	NB-ZHOUMINGYAO:0	LISTENING
TCP	192.168.79.1:139	NB-ZHOUMINGYAO:0	LISTENING
TCP	192.168.154.1:139	NB-ZHOUMINGYAO:0	LISTENING
TCP	:::135	NB-ZHOUMINGYAO:0	LISTENING
TCP	:::443	NB-ZHOUMINGYAO:0	LISTENING
TCP	:::445	NB-ZHOUMINGYAO:0	LISTENING
TCP	:::1025	NB-ZHOUMINGYAO:0	LISTENING
TCP	:::1026	NB-ZHOUMINGYAO:0	LISTENING
TCP	:::1027	NB-ZHOUMINGYAO:0	LISTENING
TCP	:::1028	NB-ZHOUMINGYAO:0	LISTENING
TCP	:::1031	NB-ZHOUMINGYAO:0	LISTENING
TCP	:::1033	NB-ZHOUMINGYAO:0	LISTENING
TCP	:::1:8307	NB-ZHOUMINGYAO:0	LISTENING
UDP	0.0.0.0:500	*:*	
UDP	0.0.0.0:4500	*:*	
UDP	0.0.0.0:5355	*:*	
UDP	0.0.0.0:9909	*:*	
UDP	0.0.0.0:17869	*:*	
UDP	0.0.0.0:17883	*:*	
UDP	0.0.0.0:18889	*:*	
UDP	0.0.0.0:50867	*:*	
UDP	127.0.0.1:40000	*:*	
UDP	127.0.0.1:60571	*:*	
UDP	192.168.79.1:137	*:*	
UDP	192.168.79.1:138	*:*	
UDP	192.168.154.1:137	*:*	
UDP	192.168.154.1:138	*:*	
UDP	:::500	*:*	
UDP	:::4500	*:*	


```
UDP      [::]:5355                *:*
```

```
UDP      [fe80::a1ac:783b:9535:919c%19]:546  *:*
```

```
UDP      [fe80::dd20:a40c:64b6:5747%20]:546  *:*
```

使用管道的方式在某些情况下显得不够灵活，因此 Java7 对进程的输入和输出处理进行了更新，增加了另外的两种处理方式。第一种是继承式，即新创建进程的输入和输出与当前的 Java 进程相同。第二种是基于文件式，即把文件作为进程输入的来源和输出的目的。下面是一个继承式的例子，其中启动的进程通过 Windows 上的 cmd 工具执行 dir 命令，通过 ProcessBuilder 类的 redirectOutput 方法把进程的输出设置为继承自父进程，运行的结果会显示在 Java 程序默认的输出控制台上。

代码清单 5-3 继承式示例

```
public void startProcessBasedonJava7() throws IOException{
    ProcessBuilder pb = new ProcessBuilder("cmd.exe", "/c", "dir");
    pb.redirectOutput(Redirect.INHERIT);
    pb.start();
}
```

程序运行输出如下所示。

代码清单 5-4 清单 5-3 运行输出

驱动器 D 中的卷没有标签。
卷的序列号是 0000-D09A

D:\Project\Java8Project 的目录

```
2015/10/06 14:31 <DIR> .
2015/10/06 14:31 <DIR> ..
2015/08/20 09:28      16,060 .classpath
2015/07/21 11:21      388 .project
2015/07/21 11:21 <DIR> .settings
2015/08/10 14:35    151,679 artoolkitplus-linux-x86.jar
2015/08/10 14:35    23,677 artoolkitplus.jar
2015/10/06 14:28 <DIR> bin
2015/08/10 14:35   7,762,368 ffmpeg-linux-x86.jar
2015/08/10 14:35   224,021 ffmpeg.jar
2015/08/10 14:35   216,396 flycapture-linux-x86.jar
2015/08/10 14:35   139,275 flycapture.jar
2015/08/10 14:35   225,614 javacpp.jar
2015/08/10 14:35   335,803 javacv.jar
2015/08/10 14:35   158,490 libdc1394-linux-x86.jar
2015/08/10 14:35    29,903 libdc1394.jar
2015/08/10 14:35   78,666 libfreenect-linux-x86.jar
2015/08/10 14:35    21,303 libfreenect.jar
2015/10/06 14:31     3,085 netstat.txt
2015/08/10 14:35   7,995,786 opencv-linux-x86.jar
2015/08/10 14:35   813,781 opencv.jar
```

2015/10/06 14:28	<DIR>	src
2015/09/01 15:32		8,088 temp
2015/08/10 14:35		15,115 videoinput.jar
19 个文件 18,219,498 字节		
5 个目录 39,639,306,240 可用字节		

如果希望把进程的输入或输出改为文件，那么可以使用 `ProcessBuilder` 类中的 `redirectInput` 和 `redirectOutput` 方法的其他重载形式。下面的例子通过一个文件来保存进程的输出内容。

代码清单 5-5 `ProcessBuilderJava7` 方式

```
public void startProcessBasedonJava7_1() throws IOException{
    ProcessBuilder pb = new ProcessBuilder("cmd.exe", "/c", "dir");
    File output = Paths.get("dir.txt").toFile();
    pb.redirectOutput(output);
    pb.start();
}
```

从 API 的角度来说，Java7 通过新增的 `ProcessBuilder.Redirect` 类对进程的输入和输出重定向方式进行了统一。`ProcessBuilder.Redirect` 类提供了两种直接使用的重定向类型，一种是 Java7 之前就有的管道式，用 `ProcessBuilder.Redirect.PIPE` 来表示。另一种是前面介绍的继承式，用 `ProcessBuilder.Redirect.INHERIT` 来表示。其余 3 种方式都是与文件相关的，在使用时都需要一个 `File` 类的对象作为参数。`ProcessBuilder.Redirect.from` 表示从一个文件中读取内容作为输入，`ProcessBuilder.Redirect.to` 表示把输出写入一个文件中，`ProcessBuilder.Redirect.appendTo` 表示把输出的内容添加到一个已有的文件中。

在通过 `ProcessBuilder` 类的 `redirectInput` 和 `redirectOutput` 方法将输入和输出重定向之后，如果新的输入源和输出目标不是默认的管道方式，那么就无法访问所创建进程的 `Process` 类的对象中的对应流。例如，通过 `redirectInput` 方法把进程的输入重定向到某个文件之后，`Process` 类的对象的 `getOutputStream` 方法返回的是一个空的输出流，调用该输出流的 `write` 方法总是会抛出 `IOException` 异常。通过 `redirectOutput` 和 `redirectError` 方法把进程的正常和错误的输出重定向到某个文件之后，`Process` 类的对象的 `getInputStream` 和 `getErrorStream` 方法返回的是一个空的输入流，调用该输入流的 `read` 方法总是会返回 -1。因此，如果在 `ProcessBuilder` 类的对象中对进程的输入或输出进行了重定向，那么相应的 `Process` 类的对象使用者一定要了解这一点，以免造成使用错误。

线程可以理解成是在进程中独立运行的子任务。比如，`QQ.exe` 运行时就是很多的子任务在同时运行。再如，好友视频线程、下载文件线程、传输数据线程、发送表情线程等，这些不同的任务或者说功能都可以同时运行，其中每一项任务完全可以理解成是线程在工作，这些线程都在后台运行着。这样做有什么优点呢？更具体来讲，使用多线程有什么优点呢？以 Windows 操作系统为例，可以最大限度地利用 CPU 的空闲时间来处理其他的任务，比如一边让操作系统处理正在由打印机打印的数据，一边使用 Word 编辑文档。CPU 在这些任务之间不停地切换，由于切换的速度非常快，给使用者的感受就是这些任务似乎在同时运行。所以使用多线程技术后，可以在同一时间内运行更多不同种类的任务。

在多核时代，使用多线程可以明显地提高系统的性能。但事实上，使用多线程的方式会额外增加系统的开销。对于单任务或者单线程的应用而言，其主要资源都消耗在任务本身。它既不需

要维护并行数据结构间的一致性状态，也不需要为线程切换和调度花费时间；但对于多线程应用来说，系统除了处理功能需求外，还需要额外维护多线程环境的特有信息。例如，线程本身的元数据、线程的调度、线程上下文的切换等。

事实上，在单核 CPU 上，采用并行算法的效率一般要低于原始的串行算法。其根本原因也在此，因此，并行计算之所以能提高系统的性能，并不是因为它“少干活”了，而是因为并行计算可以更合理地进行任务调度。因此，合理的并发才能将多核 CPU 的性能发挥到极致。

注意，Scala 和 Erlang 都采用了角色模型来进行并发编程，没有采用线程概念。围绕角色模型的创新并不仅限于语言本身，角色模型也可供 Kilim 等基于 Java 的角色框架使用。

简单地理解，如果说线程是对进程的进一步分割，那么协程是对线程的进一步分割。无论是进程、线程还是协程，在逻辑层，它们都可以对应一个任务，以执行一段逻辑代码，达到一个目标。当使用协程实现一个任务时，协程并不完全占据一个线程，当一个协程处于等待状态时，它便会把 CPU 交给该线程内的其他协程。与线程相比，协程间的切换更为轻便，因此，具有更低的操作系统成本和更高的任务并发性。

注意，协程并不被 Java 语言原生支持。在 Java 中使用协程，可以使用协程框架，Kilim 就是一个比较流行的协程框架。通过 Kilim，开发人员可以以较低的开发成本，将协程引入系统。

Kilim 是一个使用 Java 编写的库，融入了角色模型的概念。在 Kilim 中，“角色”是使用 Kilim 的 Task 类型来表示的。Task 是轻量型的线程，它们通过 Kilim 的 Mailbox 类型与其他 Task 通信。Mailbox 可以接受任何类型的“消息”。例如，Mailbox 类型接受 java.lang.Object。Task 可以发送 String 消息或者甚至自定义的消息类型，这完全取决于您自己。

在 Kilim 中，所有实体都通过方法签名捆绑在一起，如果您需要同时执行几项操作，可以在一个方法中指定该行为，扩大该方法的签名以抛出 Pausable。因此，在 Kilim 中创建并发类就像在 Java 中实现 Runnable 或扩展 Thread 一样简单。只是使用 Runnable 或 Thread 的附加实体（比如关键字 synchronized）更少了。

最后，Kilim 的魔力是由一个称为 weaver 的后期进程来实现的，该进程转换类的字节码。包含 Pausablethrows 字句的方法在运行时由一个调度程序处理，该调度程序包含在 Kilim 库中。该调度程序处理有限数量的内核线程。可以利用此工具来处理更多的轻量型线程，这可以最大限度地提高上下文切换和启动的速度。每个线程的堆栈都是自动管理的。

通过一个示例实际了解 Kilim，编写了两个角色，它们扩展自 Kilim 的 Task 类型。这些类会以一种并发方式协同工作。DeferredDivision 对象将创建一个被除数和一个除数，由于除法运算很耗资源，所以 DeferredDivision 对象将要求 Calculator 类型来处理该任务。这两个角色通过一个共享 Mailbox 实例保持通信，该实例接受一个 Calculation 类型返回。这种消息类型非常简单，通过已提供的被除数和除数，Calculator 随后将执行计算并设定相应的答案。Calculator 然后将这个 Calculation 实例传回共享 Mailbox 中。

Calculation 类是一个 JavaBean，这个类型这个类型不需要任何特殊的 Kilim 代码。

代码清单 5-6 Calculation 类

```
import java.math.BigDecimal;
```

```
public class Calculation {
    private BigDecimal dividend;
    private BigDecimal divisor;
    private BigDecimal answer;

    public Calculation(BigDecimal dividend, BigDecimal divisor) {
        super();
        this.dividend = dividend;
        this.divisor = divisor;
    }

    public BigDecimal getDividend() {
        return dividend;
    }

    public BigDecimal getDivisor() {
        return divisor;
    }

    public void setAnswer(BigDecimal ans){
        this.answer = ans;
    }

    public BigDecimal getAnswer(){
        return answer;
    }

    public String printAnswer() {
        return "The answer of " + dividend + " divided by " + divisor +
            " is " + answer;
    }
}
```

`DeferredDivision` 类中使用了特定于 `Kilim` 的类。该类执行多项操作，但总体来讲它的工作非常简单：使用随机数（类型为 `BigDecimal`）创建 `Calculation` 的实例，将它们发送到 `Calculator` 角色。而且，该类还会检查共享的 `MailBox`，以查看其中是否有任何 `Calculation`。如果检索到的一个 `Calculation` 实例有一个答案，`DeferredDivision` 将打印它。

代码清单 5-7 `DeferredDivision` 类

```
import java.math.BigDecimal;
import java.math.MathContext;
import java.util.Date;
import java.util.Random;
```



```

import kilim.Mailbox;
import kilim.Pausable;
import kilim.Task;

public class DeferredDivision extends Task {

    private Mailbox<Calculation> mailbox;

    public DeferredDivision(Mailbox<Calculation> mailbox) {
        super();
        this.mailbox = mailbox;
    }

    @Override
    public void execute() throws Pausable, Exception {
        Random numberGenerator = new Random(new Date().getTime());
        MathContext context = new MathContext(8);
        while (true) {
            System.out.println("I need to know the answer of something");
            mailbox.putnb(new Calculation(
                new BigDecimal(numberGenerator.nextDouble(), context),
                new BigDecimal(numberGenerator.nextDouble(), context)));
            Task.sleep(1000);
            Calculation answer = mailbox.getnb(); // no block
            if (answer != null && answer.getAnswer() != null) {
                System.out.println("Answer is: " + answer.printAnswer());
            }
        }
    }
}

```

从上面的代码可以看到，DeferredDivision 类扩展了 Kilim 的 Task 类型，后者实际上模仿了角色模型。注意，该类还改写了 Task 的 execute 方法，后者默认情况下抛出 Pausable。因此，execute 的操作将在 Kilim 的调度程序控制下进行。也就是说，Kilim 将确保 execute 以一种安全的方式并行地运行。在 execute 方法内部，DeferredDivision 创建 Calculation 的实例并将它们放在 Mailbox 中。它使用 putnb 方法以一种非阻塞方式完成此任务。填充 mailbox 后，DeferredDivision 进入休眠状态。注意，与处于休眠状态的内核线程不同，它是由 Kilim 托管的轻量级线程。当角色唤醒之后，像前面提到的一样，它在 mailbox 中查找任何 Calculation。此调用也是非阻塞的，这意味着 getnb 可以返回 null。如果 DeferredDivision 找到一个 Calculation 实例，并且该实例的 getAnswer 方法有一个值（也就是说，不是一个已由 Calculator 类型处理过的 Calculation 实例），它将该值打印到控制台。

Mailbox 的另一端是 Calculator。与清单 5-7 中定义的 DeferredDivision 角色类似，Calculator 也扩展了 Kilim 的 Task 并实现了 execute 方法。一定要注意两个角色都共享同一个 Mailbox 实例。它们不能与不同的 Mailbox 通信，它们需要共享一个实例。相应地，两个角色都通过它们的构造函数接受一个有类型 Mailbox。

Calculator 的 `execute` 方法与 `DeferredDivision` 的相应方法一样，不断循环查找共享 Mailbox 中的项。区别在于 Calculator 调用 `get` 方法，这是一种阻塞调用。相应地，当一条 Calculation “消息”显示时，它执行请求的除法运算。最后，Calculator 将修改的 Calculation 放回到 Mailbox 中（采用非阻塞方式），然后进入休眠状态。两个角色中的休眠调用都仅用于简化控制台的读取。

代码清单 5-8 Calculator 类

```
import java.math.RoundingMode;

import kilim.Mailbox;
import kilim.Pausable;
import kilim.Task;

public class Calculator extends Task{

    private Mailbox<Calculation> mailbox;

    public Calculator(Mailbox<Calculation> mailbox) {
        super();
        this.mailbox = mailbox;
    }

    @Override
    public void execute() throws Pausable, Exception {
        while (true) {
            Calculation calc = mailbox.get(); // blocks
            if (calc.getAnswer() == null) {
                calc.setAnswer(calc.getDividend().divide(calc.getDivisor(), 8,
                    RoundingMode.HALF_UP));
                System.out.println("Calculator determined answer");
                mailbox.putnb(calc);
            }
            Task.sleep(1000);
        }
    }
}
```

接下来我们就可以将这两个角色应用到实际中就像在 Java 代码中应用两个普通的 Thread 一样。使用同一个共享 `sharedMailbox` 实例创建并扩展两个角色实例，然后调 `start` 方法来实际设置。

代码清单 5-9 CalculateMain 类

```
import kilim.Mailbox;
import kilim.Task;

public class CalculateMain {
    public static void main(String[] args) {
        Mailbox<Calculation> sharedMailbox = new Mailbox<Calculation>();
```



```
Task deferred = new DeferredDivision(sharedMailbox);
Task calculator = new Calculator(sharedMailbox);

deferred.start();
calculator.start();

}
}
```

综合上面的这些陈述，我们可以理解清楚进程、线程、协程的关系。现代的多任务操作系统的典型特征是对进程的支持，进程是一种重量级的任务调度方式，进程创建、调度和上下文切换需要花费较多的系统资源。因此，进程的并发性是非常有限的。为了增加操作系统的并发性，人们提出了线程，线程也称为轻量级进程，它的创建和切换消耗远远低于进程，正是由于这个原因，使用线程作为并发控制的基本单元，可以使应用程序拥有更高的并发能力。与进程相比，线程是一个较为轻量级的并行程序解决方案。但是，对于高并发程序而言，线程对系统资源的占用量依然不小，这也限制了系统的并发数。为了进一步提升系统的并发数量，可以对线程进一步分割，即所谓的协程。随着应用程序日趋复杂，软件对程序并发度的要求也越来越高。在超高并发量的环境中，线程也可能会显得相对沉重。操作系统会花费较多的时间在进程或线程的切换，为了使系统能够支持更高的并行度，便有了协程的概念。

5.1.3 使用多线程的原因

现代操作系统在运行一个程序时，会为其创建一个进程。例如，启动一个 Java 程序，操作系统就会创建一个 Java 进程。现在操作系统调度的最小单元是线程，也叫轻量级进程（Light Weight Process），在一个进程里可以创建多个线程，这些线程都拥有各自的计数器、堆栈和局部变量等属性，并且能够访问共享的内存变量。处理器在这些线程上高速切换，让使用者感觉到这些线程在同时执行。

一个 Java 程序从 `main()` 方法开始执行，然后按照既定的代码逻辑执行，看似没有其他线程参与，但实际上 Java 程序天生就是多线程程序，因为执行 `main()` 方法的是一个名称为 `main` 的线程。

使用多线程的几个原因：

（1）随着处理器上的核心数量越来越多，以及超线程技术的广泛应用，现在大多数计算机都更加擅长并行计算，而处理器性能的提升方式，也从更高的主频向更多的核心发展。如何利用好处理器上的多个核心也成了现在的主要问题。

线程是大多数操作系统调度的基本单元，一个程序作为一个进程来运行，程序运行过程中能够创建多个线程，而一个线程在一个时刻只能运行在一个处理器核心上。就是说，一个单线程程序在运行时只能使用一个处理器核心，所以那么再多的处理器核心加上也无法显著提升该程序的执行效率。相反，如果该程序使用多线程技术，将计算逻辑分配到多个处理器核心上，就会显著减少程序的处理时间，并且随着更多处理器核心的加入而变得更有效率。

（2）有时我们会编写一些较为复杂的业务逻辑程序，用户从单击某个按钮开始，就要等待这些操作全部完成才能看到全部逻辑完成的结果。但是这么多业务操作，如何能够让其更快地完成呢？可以使用多线程技术，即将数据一致性不强的操作派发给其他线程处理，也可以使用消息队

列，例如生成订单快照、发送邮件等。这样做的好处是相应用户请求的线程能够尽可能快地处理完成，缩短了响应时间，提升了用户体验。

(3) Java 为多线程编程提供了良好、考究并且一致的编程模型，使开发人员能够更加专注于问题的解决，即为所遇到的问题建立合适的模型，而不是绞尽脑汁地考虑如何将其多线程化。一旦开发人员建立好了模型，稍作修改总是能够方便地影射到 Java 提供的多线程编程模型上。

开发人员经常会遇到这样的方法调用情景，调用一个方法时等待一段时间，一般来说是给定一个时间段，如果该方法能够在给定的时间段之内得到结果，那么结果将立刻返回，反之，超时返回默认结果。假设超时时间段是 T ，则可以推断出在当前时间 $\text{now}+T$ 之后就会超时。

我们假设定义如下变量：

(1) 等待持续时间： $\text{REMAINING}=T$ ；

(2) 超时时间： $\text{FUTURE}=\text{now}+T$ 。

这时仅需要 $\text{wait}(\text{REMAINING})$ 即可，在 $\text{wait}(\text{REMAINING})$ 返回之后会执行 $\text{REMAINING}=\text{FUTURE}-\text{now}$ 。如果 REMAINING 小于等于 0，表示已经超时，直接退出，否则将继续执行 $\text{wait}(\text{REMAINING})$ 。

上述描述等待超时模式的伪代码如下。

代码清单 5-10 等待超时模式的伪代码

//对当前对象加锁

```
public synchronized Object get(long mills) throws InterruptedException{
    long future = System.currentTimeMillis()+mills;
    long remaining = mills;
    //当超时大于 0 并且 result 返回值不满足要求时
    while((result == null) && remaining>0){
        wait(remaining);
        remaining = future - System.currentTimeMillis();
    }
    return result;
}
```

可以看出，等待超时模式就是在等待/通知范式基础上增加了超时控制，这使得该模式相比原有范式更具有灵活性，因为即使方法执行时间过长，也不会“永久”阻塞调用者，而是会按照调用者的要求“按时”返回。

5.1.4 线程不安全范例

所谓线程安全就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。线程不安全就是指不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据。

我们来举一个例子。类 `SimpleDateFormat` 主要负责日期的转换与格式化，但在多线程的环境中，使用此类容易造成数据转换及处理的不准确，因为 `SimpleDateFormat` 类并不是线程安全的。

下面示例将实现使用类 `SimpleDateFormat` 在多线程环境下处理日期但得出的结果却是错误的情况，这也是在多线程环境开发中很容易遇到的问题。

代码清单 5-11 类 `simpleDateThread`

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class simpleDateThread extends Thread{
    private SimpleDateFormat sdf;
    private String dateString;
    public simpleDateThread(SimpleDateFormat sdf,String dateString){
        super();
        this.sdf = sdf;
        this.dateString = dateString;
    }
    public void run(){
        try{
            Date dateRef = sdf.parse(dateString);
            String newDateString = sdf.format(dateRef).toString();
            if(!newDateString.equals(dateString)){
                System.out.println("ThreadName="+this.getName()
                    +"报错了 日期字符串:"+dateString+"转换成的日期为:"
                    +newDateString);
            }
        }catch(ParseException ex){
            ex.printStackTrace();
        }
    }
}
```

代码清单 5-12 类 `simpleDateDemo`

```
import java.text.SimpleDateFormat;

public class simpleDateDemo {
    public static void main(String[] args){
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        String[] dateStringArray = new String[]{"2000-01-01","2010-01-01",
            "2002-01-01","2003-01-01",
            "2004-01-01","2005-01-01",
            "2006-01-01","2007-01-01",
            "2008-01-01","2009-01-01"};
    };
    simpleDateThread[] threadArray = new simpleDateThread[10];
```

```

        for(int i=0;i<10;i++){
            threadArray[i] = new simpleDateThread(sdf,dateStringArray[i]);
        }
        for(int i=0;i<10;i++){
            threadArray[i].start();
        }
    }
}

```

输出结果如清单所示。

代码清单 5-13 多线程不安全输出

```

ThreadName=Thread-9 报错了 日期字符串:2009-01-01 转换成的日期为:1002-01-01
ThreadName=Thread-2 报错了 日期字符串:2002-01-01 转换成的日期为:1002-01-01

```

从打印结果来看,使用单例的 SimpleDateFormat 类在多线程的环境中处理日期,容易出现日期转换错误的情况。

如果我们新增一个类 DateTools,它里面实现静态的 parse 方法,让多线程程序调用该 parse 方法,可以解决上面出现的多线程不安全问题。

代码清单 5-14 DateTools 类

```

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateTools {
    public static Date Parse(String formatPattern,String dateString) throws
    ParseException{
        return new SimpleDateFormat(formatPattern).parse(dateString);
    }
    public static String format(String formatPattern,Date date){
        return new SimpleDateFormat(formatPattern).format(date).toString();
    }
}

```

然后把类 simpleDateThread 里面 try 块的前两行代码改为如下所示,就能解决多线程不安全问题。

代码清单 5-15 simpleDateThread

```

Date dateRef = DateTools.Parse("yyyy-MM-dd", dateString);
String newDateString = DateTools.format("yyyy-MM-dd", dateRef).toString();

```

5.1.5 重排序机制

在计算机中,软件技术和硬件技术有一个共同的目标,在不改变程序执行结果的前提下,尽可能提高并行度,重排序就是这样的一种手段。

重排序指的是编译器和处理器为了优化程序性能为对指令序列进行重新排序的一种手段。如果两个操作访问同一个变量，且这两个操作中有一个为写操作，此时这两个操作之间就存在数据依赖性。例如， $a=1$ ， $b=a$ ，这是一个写后读的示例，即在写一个变量之后，再读这个位置。只要重排序两个操作的执行顺序，程序的执行结果就会被改变。

前面提到过，编译器和处理器可能会对操作做重排序。编译器和处理器在重排序时，会遵守数据依赖性，编译器和处理器不会改变存在数据依赖关系的两个操作的执行顺序。

注意，这里所说的数据依赖性仅针对单个处理器中执行的指令序列和单个线程中执行的操作，不同处理器之间和不同线程之间的数据依赖性不被编译器和处理器考虑。**as-if-serial** 语义的意思是，不管怎么重排序（编译器和处理器为了提高并行度），（单线程）程序的执行结果不能被改变。编译器、runtime 和处理器都必须遵守 **as-if-serial** 语义。

为了遵守 **as-if-serial** 语义，编译器和处理器不会对存在数据依赖关系的操作做重排序。因为这种重排序会改变执行结果。但是，如果才做之间不存在数据依赖关系，这些操作就可能被编译器和处理器重排序。例如一个例子，计算圆面积的公式如下：

```
double pi = 3.14; //A
double r = 1.0; //B
double area = pi * r * r; //C
```

A 和 C 之间存在数据依赖关系，同时 B 和 C 之间也存在数据依赖关系。因此在最终执行的指令序列中，C 不能被重排序到 A 和 B 的前面（C 排到 A 和 B 的前面，程序的结果将会被改变）。但 A 和 B 之间没有数据依赖关系，编译器和处理器可以重排序 A 和 B 之间的执行顺序。这样 JMM 并不要求 A 一定要在 B 之前执行。JMM 仅仅要求前一个操作（执行的结果）对后一个操作可见，且前一个操作按顺序排在第二个操作之前。这里操作 A 的执行结果不需要对操作 B 可见。在这种情况下，JMM 会认为这种重排序并不非法，即 JMM 允许这种重排序。

我们来看一下重排序是否会对多线程程序产生影响，假设我们有如下一个示例代码。

代码清单 5-16 类 ReorderDemo

```
class ReorderDemo {
    int a = 0;
    boolean flag = false;

    public void writer() {
        a = 1;                //1
        flag = true;          //2
    }

    public void reader() {
        if (flag) {           //3
            int i = a * a;     //4
            .....
        }
    }
}
```

假设有两个线程 A 和 B，A 首先执行 `writer()` 方法，随后 B 线程接着执行 `reader()` 方法。线程 B 在执行操作 4 时，能否看到线程 A 在操作 1 对共享变量 `a` 的写入？注意，上面的代码中，`flag` 变量是个标记，用来标识变量 `a` 是否已被写入。

答案是不一定能够看到。由于操作 1 和操作 2 没有数据依赖关系，编译器和处理器可以对这两个操作重排序；同样，操作 3 和操作 4 没有数据依赖关系，编译器和处理器也可以对这两个操作重排序。让我们先来看看，当操作 1 和操作 2 重排序时，可能会产生什么效果？请看下面的程序执行时序图。

如图 5-1 所示，操作 1 和操作 2 做了重排序。程序执行时，线程 A 首先写标记变量 `flag`，随后线程 B 读这个变量。由于条件判断为真，线程 B 将读取变量 `a`。此时，变量 `a` 还根本没有被线程 A 写入，在这里多线程程序的语义被重排序破坏了。

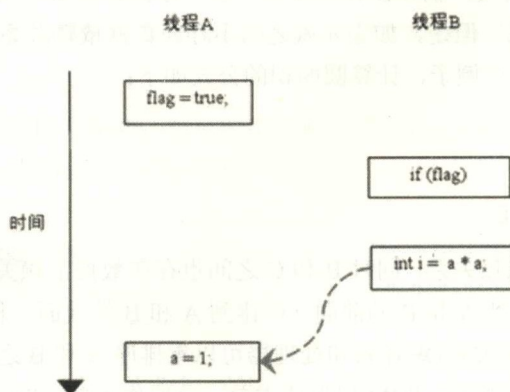


图5-1 程序执行时序图

由此，我们可以得出结论，在单线程程序中，对存在控制依赖的操作重排序，不会改变执行结果（这也是 `as-if-serial` 语义允许对存在控制依赖的操作做重排序的原因）；但在多线程程序中，对存在控制依赖的操作重排序，可能会改变程序的执行结果。

5.1.6 实例变量的数据共享

自定义线程类中的实例变量针对其他线程可以有共享和不共享两种，这在多线程之间进行交互时是很重要的一个技术点。我们首先来看一下不共享数据的情况，代码如代码清单 5-17 所示。

代码清单 5-17 不共享数据示例

```

public class unShareThread extends Thread{
    private int count = 5;

    public unShareThread(String name){
        super();
        this.setName(name); //设置线程名称
    }

    public void run(){

```



```

        super.run();
        while(count > 0){
            count--;
            System.out.println(this.currentThread().getName()+" count="+count);
        }
    }
}

```

清单 5-17 程序的运行输出如清单 5-18 所示。

代码清单 5-18 不共享数据示例运行输出

```

1, count=4
1, count=3
1, count=2
1, count=1
2, count=4
2, count=3
2, count=2
2, count=1
2, count=0
1, count=0
3, count=4
3, count=3
3, count=2
3, count=1
3, count=0

```

从上面输出可以看到，程序当中一共创建了 3 个线程，每个线程都有各自的 count 变量，自己减少自己的 count 变量的值。这样的情况就是变量不共享，这个例子当中并不存在多个线程访问同一个实例变量的情况。

共享数据的情况就是多个线程可以访问同一个变量，比如，实现投票功能的软件，多个线程可以同时处理同一个人的票数。代码如代码清单 5-19 所示。

代码清单 5-19 共享数据示例

```

public class sharedThread extends Thread{
    private int count = 5;
    public void run(){
        super.run();
        count--;
        System.out.println(this.currentThread().getName()+"count="+count);
    }
}

public class runClass {
    public static void main(String[] args){
        sharedThread mythread = new sharedThread();
    }
}

```

```

Thread aa = new Thread(mythread, "aa");
Thread bb = new Thread(mythread, "bb");
Thread cc = new Thread(mythread, "cc");
aa.start();
bb.start();
cc.start();
    }
}

```

运行代码后输出为 aacount=4 bbcount=3 cccount=2。可以看到，线程 aa,bb,cc 同时都对 count 进行处理，产生了“非线程安全”问题。在某些 JVM 中，i++ 的操作要分成 3 个步骤：

- (1) 取得原有 i 值。
- (2) 计算 i+1。
- (3) 对 i 赋值。

在这 3 个步骤中，如果有多个线程同时访问，那么一定会出现非线程安全问题。

可以通过加 synchronized 的方式让多个线程之间进行同步，也就是按顺序排队的方式减 1。

上面的例子中通过在 run 方法前加上 synchronized 关键字，使多个线程在执行 run 方法时，以排队的方式进行处理。当一个线程调用 run 前，先判断 run 方法有没有被上锁，如果上锁，说明有其他线程正在调用 run 方法，必须等其他线程调用结束后才可以执行 run 方法。synchronized 可以在任意对象及方法上加锁，而加锁的这段代码称为“互斥区”或“临界区”。

当一个线程想要执行同步方法里面的代码时，线程首先尝试去拿这把锁，如果能够拿到锁，那么该线程就可以执行 synchronized 里面的代码。如果拿不到锁，那么这个线程就会不断尝试拿这把锁，直到能够拿到为止，而且是有多个线程同时去争夺这把锁。

非线程安全指的是多个线程对同一个对象中的同一个实例变量进行操作时会出现值被更改、值不同步的情况，进而影响程序的执行流程，即产生所谓的“脏读”。

线程安全指的是同样情况下，获得实例变量的值是经过同步处理的，不会出现脏读的现象。

非线程安全问题存在于实例变量中，如果是方法内部的私有变量，则不存在非线程安全问题，所得结果也就是线程安全的了。

5.1.7 生产者与消费者模式

生产者消费者问题是研究多线程程序时绕不开的经典问题之一，我们可以把整个问题的场景描述为有一块缓冲区作为仓库，生产者可以将产品放入仓库，消费者则可以从仓库中取走产品。解决生产者/消费者问题的方法可分为两类：

- (1) 采用某种机制保护生产者和消费者之间的同步；
- (2) 在生产者和消费者之间建立一个管道。

第一种方式有较高的效率，并且易于实现，代码的可控制性较好，属于常用的模式。第二种管道缓冲区不易控制，被传输数据对象不易于封装等，实用性不强。

同步问题核心在于：如何保证同一资源被多个线程并发访问时的完整性。常用的同步方法是

采用信号或加锁机制，保证资源在任意时刻至多被一个线程访问。Java 语言在多线程编程上实现了完全对象化，提供了对同步机制的良好支持。在 Java 中一共有四种方法支持同步，其中前三个是同步方法，一个是管道方法。这里我们介绍前三个方法。

(1) wait()/notify()方法

(2) await()/signal()方法

(3) BlockingQueue 阻塞队列方法

(4) PipedInputStream/PipedOutputStream

wait()/notify()方法

wait()/notify()方法是基类 Object 的两个方法，也就意味着所有 Java 类都会拥有这两个方法，这样，我们就可以为任何对象实现同步机制。

- wait()方法：当缓冲区已满/空时，生产者/消费者线程停止自己的执行，放弃锁，使自己处于等待状态，让其他线程执行。
- notify()方法：当生产者/消费者向缓冲区放入/取出一个产品时，向其他等待的线程发出可执行的通知，同时放弃锁，使自己处于等待状态。

await()/signal()方法

在 JDK5.0 之后，Java 提供了更加健壮的线程处理机制，包括同步、锁定、线程池等，它们可以实现更细粒度的线程控制。await()和 signal()就是其中用来做同步的两种方法，它们的功能基本上和 wait()/notify()相同，完全可以取代它们，但是它们和新引入的锁定机制 Lock 直接挂钩，具有更大的灵活性。通过在 Lock 对象上调用 newCondition()方法，将条件变量和一个锁对象进行绑定，进而控制并发程序访问竞争资源的安全。

BlockingQueue 阻塞队列方法

生产者-消费者设计是围绕阻塞队列展开的，生产者把数据放入队列，并使数据可用，当消费者为适当的行为做准备时会从队列中获取数据。生产者不需要知道消费者的省份或者数量，甚至根本没有消费者——它们只负责把数据放入队列。类似地，消费者也不需要知道生产者是谁，以及是谁给它们安排的工作。BlockingQueue 可以使用任意数量的生产者和消费者，从而简化了生产者-消费者设计的实现。最常见的生产者-消费者设计是将线程池与工作队列相结合。

阻塞队列支持生产者-消费者设计模式。一个生产者-消费者设计分离了“生产产品”和“消费产品”。该模式不会发现一个工作便立即处理，而是把工作置于一个任务（“to do”）清单中，以备后期处理。生产者-消费者模式简化了开发，因为它解除了生产者和消费者之间相互依赖的代码。生产者和消费者以不同的或者变化的速度生产和消费数据，生产者-消费者模式将这些活动解耦，因而简化了工作负荷的管理。

阻塞队列简化了消费者的编码，因为程序代码会保持阻塞直到可用数据出现。如果生产者不能足够快地产生工作，让消费者忙碌起来，那么消费者只能一直等待，直到有工作可做。同时，put 方法的阻塞特性也大大地简化了生产者的编码；如果使用一个有界队列，那么当队列充满的时候，生产者就会阻塞，暂不能生成更多的工作，从而给消费者时间来赶进度。

虽然生产者-消费者模式可以把生产者和消费者的代码相互解耦合，但是它们的行为还是间接地通过共享队列耦合在一起了。

类库中包含一些 `BlockingQueue` 的实现，其中 `LinkedBlockingQueue` 和 `ArrayBlockingQueue` 是 FIFO 队列，与 `LinkedList` 和 `ArrayList` 相似，但是却拥有比同步 `List` 更好的并发性能。`PriorityBlockingQueue` 是一个按优先级顺序排序的队列，当你不希望按照 FIFO 的属性处理元素时，这个 `PriorityBlockingQueue` 是非常有用的。正如其他排序的容器一样，`PriorityBlockingQueue` 可以比较元素本身的自然顺序（如果它们实现了 `Comparable`），也可以使用一个 `Comparator` 进行排序。

基于 `BlockingQueue` 实现的 `SynchronousQueue`，它根本上不是一个真正的队列，因为它不会为队列元素维护任何存储空间。不过，它维护一个排队的线程清单，这些线程等待把元素加入（`enqueue`）队列或者移出（`dequeue`）队列。因为 `SynchronousQueue` 没有存储能力，所以除非另一个线程已经准备好参与移交工作，否则 `put` 和 `take` 会一直阻止。`SynchronousQueue` 这类队列只有在消费者充足的时候比较合适，它们总能为下一个任务做好准备。关于这方面知识，我们会在后面详细讲解。

生产者-消费者模式同样带来了一些性能方面的提高。生产者和消费者可以并发地执行，如果一个受限于 I/O，另一个受限于 CPU，那么并发执行的全部产出会高于顺序执行的产出。

5.1.8 线程池的使用

多线程的软件设计方法确实可以最大限度地发挥现代多核处理器的计算能力，提高生产系统的吞吐量和性能。但是，若不加控制和管理地随意使用线程，对系统的性能反而会产生不利的影响。

为了给并行程序开发提供更好的支持，Java 不仅提供了 `Thread` 类、`Runnable` 类接口等简单的多线程支持工具，为了改善并发程序性能，在 JDK 中还提供了用于多线程管理的线程池。

第 4 章已经初步介绍了线程池概念。从内部实现上看，线程池技术可主要划分为工作者线程、待处理任务存储线程、线程池初始化、处理业务任务算法、工作者的增减算法、线程池终止等 6 大技术点。

- **工作者线程 worker**：即线程池中可以重复利用起来执行任务的线程，一个 worker 的生命周期内会不停的处理多个业务 job。线程池“复用”的本质就是复用一个 worker 去处理多个 job，“流控”的本质就是通过 worker 数量的控制实现并发数的控制。通过设置不同的参数来控制 worker 的数量可以实现线程池的容量伸缩从而实现复杂的业务需求
- **待处理工作 job 的存储队列**：工作者线程 workers 的数量是有限的，同一时间最多只能处理最多 workers 数量个 job。对于来不及处理的 job 需要保存到等待队列里，空闲的工作者 work 会不停的读取空闲队列里的 job 进行处理。基于不同的队列实现，可以扩展出多种功能的线程池，如定制队列出队顺序实现带处理优先级的线程池、定制队列为阻塞有界队列实现可阻塞能力的线程池等。流控一方面通过控制 worker 数控制并发数和处理能力，一方面可基于队列控制线程池处理能力的上限。
- **线程池初始化**：即线程池参数的设定和多个工作者 workers 的初始化。通常有一开始就初始化指定数量的 workers 或者有请求时逐步初始化工作者两种方式。前者线程池启动初期

响应会比较快但造成了空载时的少量性能浪费，后者是基于请求量灵活扩容但牺牲了线程池启动初期性能达不到最优。

- **处理业务 job 算法：**业务给线程池添加任务 job 时线程池的处理算法。有的线程池基于算法识别直接处理 job 还是增加工作者数处理 job 或者放入待处理队列，也有的线程池会直接将 job 放入待处理队列，等待工作者 worker 去取出执行。
- **workers 的增减算法：**业务线程数不是持久不变的，有高低峰期。线程池要有自己的算法根据业务请求频率高低调节自身工作者 workers 的数量来调节线程池大小，从而实现业务高峰期增加工作者数量提高响应速度，而业务低峰期减少工作者数来节省服务器资源。增加算法通常基于几个维度进行：待处理工作 job 数、线程池定义的最大最小工作者数、工作者闲置时间。
- **线程池终止逻辑：**应用停止时线程池要有自身的停止逻辑，保证所有 job 都得到执行或者抛弃。

一种最为简单的线程创建和回收的方法类似如代码清单 5-20 所示。

代码清单 5-20 最简单的线程操作方法

```
new Thread(new Runnable() {
    @Override
    public void run() {
        //do something
    }
}).start();
```

以上代码创建了一个线程，并在 run()方法结束后，自动回收该线程。在简单的应用系统中，这段代码并没有太多问题。但是，在真实的生产环境中，系统由于真实环境的需要，可能会开启很多线程来支撑其应用。而当线程数量过大时，反而会耗尽 CPU 和内存资源。

首先，虽然与进程相比，线程是一种轻量级的工具，但其创建和关闭依然需要花费时间，如果为每一个小的任务都创建一个线程，很有可能出现创建和销毁线程所占用的时间大于该线程真实工作所消耗的时间，反而会得不偿失。

其次，线程本身也是要占用内存资源的，大量的线程会抢夺资源，如果处理不当，可能会导致 Out of Memory 异常。即便没有，大量的线程回收也会给 GC 带来很大的压力，延长 GC 的停顿时间。

因此，对线程使用必须掌握一个度，在有限的范围内，增加线程的数量可以明显提高系统的吞吐量，但一旦超出了这个范围，大量的线程只会拖垮应用系统。因此，在生产环境中使用线程，必须对其加以控制和管理，否则盲目地大量创建线程对系统性能是有伤害的。

为了节省系统在多线程并发时不断创建和销毁线程所带来的额外开销，我们需要引入线程池。线程池的基本功能就是进行线程的复用。当系统接受一个提交的任务，需要一个线程时，并不急着立即去创建线程，而是先去线程池查找是否有空余的线程，若有，则直接使用线程池中的线程工作，若没有，再去创建新的线程。待任务完成后，也不简单地销毁线程，而是将线程放入线程池的空闲队列，等待下次使用时再次调用。这样可以在线程频繁调度的场合，节约不少系统开销，

即创建和销毁线程的开销。

下面的例子给出一个最为简单的线程池实现，该实现不是一个完善的线程池，但已经使用最简单的代码实现了一个基本线程池的核心功能，有助于读者快速理解线程池的实现，并创建自己的线程池。

首先是线程池的实现，如代码清单 5-21 所示。

代码清单 5-21 线程池的实现

```
import java.util.List;
import java.util.Vector;

public class ThreadPool {
    private static ThreadPool instance = null;
    //空闲的线程队列
    private List<PThread> idleThreads;
    //已有的线程总数
    private int threadCounter;
    private boolean isShutdown = false;

    private ThreadPool(){
        this.idleThreads = new Vector(5);
        threadCounter = 0;
    }

    public int getCreatedThreadsCount(){
        return threadCounter;
    }

    //取得线程池的实例
    public synchronized static ThreadPool getInstance(){
        if(instance == null){
            instance = new ThreadPool();
        }
        return instance;
    }

    //将线程放入池中
    protected synchronized void repool(PThread repoolingThread){
        if(!isShutdown){
            idleThreads.add(repoolingThread);
        }else{
            repoolingThread.shutdown();
        }
    }

    //停止池中所有线程
    public synchronized void shutdown(){
```



```

        isShutdown = true;
        for(int threadIndex = 0; threadIndex < idleThreads.size(); threadIndex++){
            PThread idleThread = (PThread)idleThreads.get(threadIndex);
            idleThread.shutdown();
        }
    }

    //执行任务
    public synchronized void start(Runnable target){
        PThread thread = null;
        //如果有空闲线程, 则直接使用
        if(idleThreads.size() > 0){
            int lastIndex = idleThreads.size() - 1;
            thread = (PThread)idleThreads.get(lastIndex);
            idleThreads.remove(lastIndex);
            //立即执行这个任务
            thread.setTarget(target);
        }else{
            //没有空闲线程
            threadCounter++;
            thread = new PThread(target, "PThread #" + threadCounter, this);
            //启动这个线程
            thread.start();
        }
    }
}

```

要使用上述线程池, 需要一个永不退出的线程与之配合。PThread 就是这样一个线程, 它的线程主体部分是一个无限循环, 该线程在手动关闭前永不结束, 并一直等待新的任务达到。代码如下代码清单 5-22 所示。

代码清单 5-22 线程守护示例

```

public class PThread extends Thread{
    //线程池
    private ThreadPool pool;
    //任务
    private Runnable target;
    private boolean isShutdown = false;
    private boolean isIdle = false;
    //构造函数
    public PThread(Runnable target, String name, ThreadPool pool){
        super(name);
        this.pool = pool;
        this.target = target;
    }

    public Runnable getTarget(){
        return target;
    }
}

```

```

    }

    public boolean isIdle(){
        return isIdle;
    }

    public void run(){
        //只要没有关闭，则一直不结束该线程
        while(!isShutdown){
            isIdle = false;
            if(target != null){
                //运行任务
                target.run();
            }
            //任务结束了，到闲置状态
            isIdle = true;
            try{
                //该任务结束后，不关闭线程，而是放入线程池空闲队列
                pool.repool(this);
                synchronized(this){
                    //线程空闲，等待新的任务到来
                    wait();
                }
            }catch(InterruptedException ex){
                ex.printStackTrace();
            }
            isIdle = false;
        }
    }

    public synchronized void setTarget(java.lang.Runnable newTarget){
        target = newTarget;
        //设置了任务之后，通知 run 方法，开始执行这个任务
        notifyAll();
    }

    //关闭线程
    public synchronized void shutDown(){
        isShutDown = true;
        notifyAll();
    }
}

```

使用线程池后，线程的创建和关闭通常由线程池维护。线程通常不会因为执行完一次任务而被关闭，线程池中的线程会被多个任务重复使用。

首先定义一个线程类，作为任务对象，如代码清单 5-23 所示。

代码清单 5-23 定义一个线程类

```

public class MyThread implements Runnable{
    protected String name;

    public MyThread(){

    }

    public MyThread(String name){
        this.name = name;
    }

    @Override
    public void run() {
        // TODO Auto-generated method stub
        try{
            Thread.sleep(1000);
        }catch (InterruptedException ex){
            ex.printStackTrace();
        }
    }
}

```

我们尝试创建 10 万个线程，采用不用线程池和使用线程池两种不同的方法，代码如代码清单 5-24 所示。

代码清单 5-24 两种不同方法创建线程

```

public class TestMyThread {
    public static void main(String[] args){
        long start = System.currentTimeMillis();
        for(int i=0;i<100000;i++){
            new Thread(new MyThread("testNoThreadPool"+Integer.toString(i))).start();
        }
        System.out.println(System.currentTimeMillis() - start);

        start = System.currentTimeMillis();
        for(int i=0;i<100000;i++){
            ThreadPool.getInstance().start(new MyThread("testNoThreadPool"+Integer.
toString(i)));
        }
        System.out.println(System.currentTimeMillis() - start);
    }
}

```

输出结果分别为 2465ms 和 734ms，结果表明，为实现线程池的调度，花费时间较长。在未使用线程池的实现中，实际产生线程数量 10 万个，但在使用线程池的实现中，由于任务一边被调度，一边被执行，在整个任务调度过程中，有部分线程因为执行完毕，被重新返回线程池内，所以这

些线程被复用，实际产生的线程数量小于 10 万个。由此可见，在线程频繁被调度时，复用线程，对提高系统性能非常有帮助。

我们来考虑一个具体的实例。对于服务端的程序，经常面对的是客户端传入的短小（执行时间短、工作内容较为单一）任务，需要服务端快速处理并返回结果。如果服务端每次接受到一个任务，创建一个线程，然后进行执行，这在原型阶段是个不错的选择，但是面对成千上万的任务递交进服务器时，如果还是采用一个任务一个线程的方式，那么将会创建数以万计的线程，这不是一个好的选择。因为这会使操作系统频繁的进行线程上下文切换，无故增加系统的负载，而线程的创建和消亡都是需要消耗系统资源的，也无疑浪费了系统资源。

线程池技术能够很好解决这个问题，它预先创建了若干数量的线程，并且不能由用户直接对线程的创建进行控制，在这个前提下重复使用固定或较为固定数目的线程来完成任务的执行。这样做的好处是，一方面，消除了频繁创建和消亡线程的系统资源开销，另一方面，面对过量任务的提交能够平缓的劣化。

为了更好地控制多线程，JDK 提供了一套 `Executors` 框架，帮助开发人员有效地进行线程控制。框架为了 `java.util.concurrent` 包中，是 JDK 开发包的核心类。其中 `ThreadPoolExecutor` 表示一个线程池，当然它的实现比上文中的 `ThreadPool` 复杂许多。`Executors` 类则扮演者线程池工厂的角色，通过 `Executors` 可以取得一个特定功能的线程池。`ThreadPoolExecutor` 类实现了 `Executor` 接口，因此通过这个接口，任何 `Runnable` 的对象都可以被 `ThreadPoolExecutor` 调度。

注意，如果没有特殊要求，可以直接使用 JDK 中的内置线程池来改善系统的性能。

线程池的大小对系统的性能有一定的影响。过大或者过小的线程数量都无法发挥最优的系统性能，但是线程池大小的确定也不需要做得非常精确，因为只要避免极大和极小的两个极端情况发生就可以了，线程池的大小对系统的性能并不会影响太大。一般来说，确定线程池的大小需要考虑 CPU 数量、内存大小、JDBC 连接等诸多因素。一个估算线程池大小的经验公式：

$N_{cpu} = \text{CPU 的数量}$

$U_{cpu} = \text{目标 CPU 的使用率, } U_{cpu} \text{ 的值在 } 0-1 \text{ 之间}$

$W/C = \text{等待时间与计算时间的比率}$

为保持处理器达到期望的使用率，最优的线程池的大小等于 $N_{threads} = N_{cpu} * U_{cpu} * (1 + W/C)$ 。

5.2 锁机制对比

5.2.1 锁机制概述

锁是用来控制多个线程访问共享资源的方式。一般来说，一个锁能够防止多个线程同时访问共享资源（但是有些锁可以允许多个线程并发的访问共享资源，例如读写锁）。在 `Lock` 接口出现之前，Java 程序是靠 `synchronized` 关键字实现锁功能的，而 Java5 之后，并发包中新增了 `Lock` 接口用来实现锁功能，它提供了与 `synchronized` 关键字类似的同步功能，只是在使用时需要显式地获取和释放锁。虽然它却少了（通过 `synchronized` 块或者方法所提供的）隐式获取释放锁的便捷性，但是却拥有了锁获取与释放的可操作性、可中断的获取锁以及超时获取锁等多种 `synchronized`

关键字所不具备的同步特性。

使用 `synchronized` 关键字将会隐式地获取锁，但是它将锁的获取和释放固化了，也就是先获取再释放。当然，这种方式简化了同步的管理，可是扩展性没有显示的锁获取和释放来得好。举一个例子，针对一个场景，我们手把手地进行锁获取和释放，先获得锁 A，然后再释放锁 B，当锁 B 获得后，释放锁 A 同时获取锁 C，当锁 C 获得后，再释放锁 B 同时获取锁 D，以此类推。这种场景下，`synchronized` 关键字就不那么容易实现了，而使用 `Lock` 却容易许多。

`Lock` 的使用很简单，如下面代码所示，后续章节会详细介绍各种 `Lock` 锁的实现类。

代码清单 5-25 Lock 锁基本实现

```
Lock lock = new ReentrantLock();
lock.lock();
try{
}finally{
    lock.unlock();
}
```

在 `finally` 块中释放锁，目的是保证在获取到锁之后，最终能够被释放。注意，不要把获取锁的过程写在 `try` 块中，因为如果在获取锁（自定义锁的实现）时发生了异常，异常抛出的同时也会导致锁无故释放。

相比较 `synchronized` 关键字而言，`Lock` 锁有如下一些特点。

- (1) **尝试非阻塞地获取锁**：当前线程尝试获取锁，如果这一时刻没有被其他线程获取到，则同时获取并持有锁；
- (2) **能被中断地获取锁**：与 `synchronized` 关键字不同，获取到锁的线程能够响应终端，当获取到锁的线程被中断时，中断异常将会被抛出，同时锁会被释放；
- (3) **超时获取锁**：在制定的截止时间之前获取锁，如果截止时间到了仍然无法获取锁，则返回。

业务逻辑的实现过程中，往往需要保证数据访问的排他性。如在金融系统的日终结算处理中，我们希望针对某个 `cut-off` 时间点的数据进行处理，而不希望在结算进行过程中（可能是几秒，也可能是几个小时），数据再发生变化。此时，我们就需要通过一些机制来保证这些数据在某个操作过程中不会被外界修改，这样的机制，也就是所谓的“锁”，给我们选定的目标数据上锁，使其无法被其他程序修改。

悲观锁（`Pessimistic Locking`），正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度。因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

乐观锁（`Optimistic Locking`），相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。如一个金融系统，当

某个操作员读取用户的数据，并在读出的用户数据的基础上进行修改时（如更改用户账户余额），如果采用悲观锁机制，也就意味着整个操作过程中（从操作员读出数据、开始修改直至提交修改结果的全过程，甚至还包括操作员中途去煮咖啡的时间），数据库记录始终处于加锁状态，可以想见，如果面对几百上千个并发，这样的情况将导致怎样的后果。

5.2.2 Synchronized 使用技巧

关键字 `synchronized` 拥有可重入锁的功能，也就是在使用 `synchronized` 时，当一个线程得到一个对象锁后，再次请求此对象锁时是可以再次得到该对象的锁的。这也证明在一个 `synchronized` 方法/块的内部调用本类的其他 `synchronized` 方法/块时，是永远可以得到锁的。`synchronized` 用的锁是存在 Java 对象头里的。如果对象是数组类型，则虚拟机用 3 个字宽存储对象头，如果对象是非数组类型，则用 2 个字宽存储对象头。在 32 位虚拟机中，1 字宽等于 4 字节，即 32Bit。

我们首先来看一个 `Synchronized` 示例，如代码清单 5-26 所示。

代码清单 5-26 Synchronized 示例

```
public class MyObject {
    public void methodA(){
        try{
            System.out.println("begin methodA threadName="
                               +Thread.currentThread().getName());
            Thread.sleep(500);
            System.out.println("end");
        }catch(InterruptedException ex){
            ex.printStackTrace();
        }
    }
}
```

```
public class ThreadA extends Thread{
    private MyObject object;
    public ThreadA(MyObject object){
        super();
        this.object = object;
    }
    @Override
    public void run(){
        super.run();
        object.methodA();
    }
}
```

```
public class ThreadB extends Thread{
    private MyObject object;
    public ThreadB(MyObject object){
```



```

        super();
        this.object = object;
    }
    @Override
    public void run(){
        super.run();
        object.methodA();
    }
}

```

```

public class runClass {
    public static void main(String[] args){
        MyObject object = new MyObject();
        ThreadA a = new ThreadA(object);
        a.setName("A");
        ThreadB b = new ThreadB(object);
        b.setName("B");
        a.start();
        b.start();
    }
}

```

如果具体执行类 `MyObject` 的 `methodA` 方法不加上关键字 `synchronized`, 那么输出如代码清单 5-27 所示。

代码清单 5-27 运行输出 1

```

begin methodA threadName=B
begin methodA threadName=A
end
end

```

如果加上关键字 `synchronized`, 输出会如清单 5-28 所示。

代码清单 5-28 运行输出 2

```

begin methodA threadName=A
end
begin methodA threadName=B
end

```

通过下面的实例我们可以看到, 调用关键字 `synchronized` 声明的方法一定是排队运行的。另外只有共享资源的读写访问才需要同步化, 如果不是共享资源, 那么根本就没有同步的必要。

我们知道, 关键字 `synchronized` 取得的锁都是对象锁, 而不是把一段代码或方法(函数)当作锁, 所以在下面的示例中, 哪个线程先执行带 `synchronized` 关键字的方法, 哪个线程就持有该方法所属对象的锁 `Lock`, 那么其他线程只能呈等待状态, 前提是多个线程访问的是同一个对象。

但如果多个线程访问多个对象, 则 JVM 会创建多个锁, 下面的示例就是创建了 2 个

sychrnoizedFunc.java 类的对象，所以就产生了 2 个锁。

代码清单 5-29 创建多个锁示例

```
public class sychrnoizedFunc {
    private int num = 0;
    //同步方法，说明此方法应该被顺序调用
    synchronized public void addI(String username){
        try{
            if(username.equals("a")){
                num = 100;
                System.out.println("a set over");
                Thread.sleep(12000);
            }else{
                num = 200;
                System.out.println("b set over");
            }
            System.out.println(username + "num=" + num);
        }catch(Exception ex){
            ex.printStackTrace();//TODO auto-generated catch block
        }
    }
}

public class sychrnoizedFuncRun {
    public static void main(String[] args){
        sychrnoizedFunc numRef1 = new sychrnoizedFunc();
        sychrnoizedFunc numRef2 = new sychrnoizedFunc();
        ThreadA athread = new ThreadA(numRef1);
        athread.start();
        ThreadB bthread = new ThreadB(numRef1);
        bthread.start();
    }
}

public class ThreadA extends Thread{
    private sychrnoizedFunc numRef;

    public ThreadA(sychrnoizedFunc numRef){
        super();
        this.numRef = numRef;
    }

    @Override
    public void run(){
        super.run();
        numRef.addI("a");
    }
}

public class ThreadB extends Thread{
    private sychrnoizedFunc numRef;
```



```

public ThreadB(sychronoziedFunc numRef){
    super();
    this.numRef = numRef;
}

@Override
public void run(){
    super.run();
    numRef.addI("b");
}
}

```

运行输出如清单 5-30 所示。

代码清单 5-30 代码 5-29 运行输出

```

a set over
anum=100
b set over
bnum=200

```

上面代码中两个线程分别访问以同一个类的两个不同实例的相同名称的同步方法。

虽然 `synchronized` 功能强大,但是使用关键字 `synchronized` 声明方法在某些情况下是有弊端的,比如线程 A 调用同步方法执行一个长时间任务,那么 B 线程则必须等待比较长时间。在这样的情况下可以使用 `synchronized` 同步语句块来解决。当两个并发线程访问同一个对象 `object` 中的 `synchronized(this)` 同步代码块时,一段时间内只能有一个线程被执行,另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。

代码清单 5-31 对象等待示例

```

public class ObjectService {
    public void serviceMethod(){
        try{
            synchronized(this){
                System.out.println("begin time="+System.currentTimeMillis());
                Thread.sleep(2000);
                System.out.println("end time="+System.currentTimeMillis());
            }
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}

public class ThreadA extends Thread{
    private ObjectService service;
    public ThreadA(ObjectService service){

```

```

        super();
        this.service = service;
    }
    @Override
    public void run(){
        service.serviceMethod();
    }
}

public class ThreadB extends Thread{
    private ObjectService service;
    public ThreadB(ObjectService service){
        super();
        this.service = service;
    }
    @Override
    public void run(){
        service.serviceMethod();
    }
}

public class runClass {
    public static void main(String[] args){
        ObjectService service = new ObjectService();
        ThreadA a = new ThreadA(service);
        a.setName("a");
        a.start();
        ThreadB b = new ThreadB(service);
        b.setName("b");
        b.start();
    }
}

```

上面的实验中虽然使用了 `synchronized` 同步代码块，但执行的效率还是没有提高，执行的效果还是同步运行的。在使用同步 `synchronized(this)` 代码块时需要注意的是，当一个线程访问 `object` 的一个 `synchronized(this)` 同步代码块时，其他线程对同一个 `object` 中所有其他 `synchronized(this)` 同步代码块的访问将被阻塞，这说明 `synchronized` 使用的是一个对象监视器，锁定了当前对象。

类似于 `synchronized` 机制，JVM 基于进入和退出 `Monitor` 对象来实现方法同步和代码块同步。代码块同步是使用 `monitorenter` 和 `monitorexit` 指令实现的。`monitorenter` 指令是在编译后插入到同步代码块的开始位置，而 `monitorexit` 指令是插入到方法结束处和异常处，JVM 要保证每个 `monitorenter` 必须有对应的 `monitorexit` 与之配对。任何对象都有一个 `monitor` 与之关联，如果一个 `monitor` 被持有后，它将处于锁定状态。线程执行到 `monitorenter` 指令时，将会尝试获取对象所对应的 `monitor` 的所有权，即尝试获得对象的锁。

5.2.3 Volatile 的使用技巧

Java 对象中声明为 `volatile` 的字段常用于线程之间状态信息的同步，即线程从对象中读取的 `volatile` 字段值就是上次写入该 `volatile` 变量中的值，不论该线程正在读还是写，也不论这些线程在什么地方运行，它们可以在不同的 CPU 插槽上，即不同的 CPU Socket 上，或者在不同的 CPU 核上。使用 `volatile` 也会带来一些副作用，它会限制现代 JVM 的 JIT 编译器对这个字段的优化，比如 `volatile` 字段必须遵守一定的指令规则。

简而言之，`volatile` 字段值在应用程序的所有线程和 CPU 缓存中必须保持同步。例如，存放在 CPU 缓存中的 `volatile` 字段值被一个线程修改后，存有该 `volatile` 变量原始值在其他 CPU 的缓存中的县城，在线程读取本地 CPU 缓存中 `volatile` 字段之前必须对其更新，否则就只能强制制定从内存中读取更新过的 `volatile` 字段值。为了确保 CPU 缓存及时更新，即在各个线程之间保持同步，出现 `volatile` 字段的地方都会加入一条 CPU 指令，即内存屏障，通常称为 `membar` 或 `fence`，一旦 `volatile` 字段值发生变化就会触发 CPU 缓存更新。

对一个拥有多 CPU 缓存，性能要求很高的应用程序，频繁更新 `volatile` 字段可能导致性能问题。然而，实际上很少会有 Java 应用程序依赖频繁更新的 `volatile` 字段，但是总有一些例外发生。如果你留意一点，即频繁更新、改变或写入 `volatile` 字段有可能导致性能问题（读取 `volatile` 字段不会造成性能问题），就不太容易碰到这类问题了。

如果你观察到 `volatile` 字段上存在大量的 CPU 高速缓存未命中并且分析源码后发现对 `volatile` 字段频繁的写操作，基本可以断定应用程序的性能问题源于不恰当地使用了 `volatile` 变量。这种情况的解决方案是尽量减少对 `volatile` 变量的写操作，或者对应用程序进行重构避免使用 `volatile` 字段。不要直接删除 `volatile` 字段，这可能会破坏程序的正确性或引入潜在的竞争条件。一个性能稍差的应用程序比一个错误的实现或有潜在竞争问题的程序要好得多。

`volatile` 是轻量级的 `synchronized`，它在多处理器开发中保证了共享变量的“可见性”。可见性的意思是当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值，即如果一个字段被声明为 `volatile`，Java 线程内存模型确保所有线程看到这个变量的值是一致的。如果 `volatile` 变量修饰符使用恰当的话，它比 `synchronized` 的使用和执行成本更低，因为它不会引起线程上下文的切换和调度。

前面了解了这么多基础知识，那么 `Volatile` 是如何来保证可见性的呢？

为了提高处理速度，处理器不会直接和内存进行通信，而是先将系统内存的数据读到内存缓存（L1、L2 或其他）后再进行操作，但操作完不知道何时会写回内存。如果对声明了 `volatile` 的变量进行写操作，JVM 就会向处理器发送一条 `Lock` 前缀的指令，将这个变量所在缓存行的数据写回到系统内存。但是，就算写回到内存，如果其他处理器缓存的值还是旧的，再执行计算操作就会有问题。所以，在多处理器下，为了保证各个处理器的缓存是一致的，就会实现缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置或无效状态，当处理器对这个数据进行修改操作的时候，会重新从系统内存中把数据读到处理器缓存里。

底层汇编代码会针对 `Volatile` 关键字修饰的变量添加 `Lock` 前缀的指令，该指令在多核处理器下会做两件事情：

- (1) 将当前处理器缓存行的数据写回到系统内存。Lock 前缀指令导致在执行指令期间，声明处理器的 LOCK#信号。在多处理器环境中，LOCK#信号确保在声明该信号期间，处理器可以独占任何共享内存。但是，再最近的处理器里，LOCK#信号一般不锁总线，而是锁缓存，毕竟锁总线开销比较大。
- (2) 这个写回内存的操作会使在其他 CPU 里缓存了该内存地址的数据无效。在多核处理器系统中进行操作的时候，IA-32 和 Intel64 处理器能嗅探其他处理器访问系统内存和它们的内存缓存。处理器使用嗅探技术保证它的内部缓存、系统内存和其他处理器的缓存的数据在总线上保持一致。

5.2.4 队列同步器

队列同步器 (AbstractQueuedSynchronizer)，是用来构建锁或者其他同步组件的基础框架，它使用了一个 int 成员变量表示同时状态，通过内置的 FIFO 队列来完成资源获取线程的排队工作，它是实现大部分同步需求的基础。

队列同步器的主要使用方式是继承，子类通过继承同步器并实现它的抽象方法来管理同步状态，在抽象方法的实现过程中免不了要对同步状态进行更改，这时就需要使用同步器提供的 3 个方法，getState()、setState(int newState)和 compareAndSetState(int expect,int update)来进行操作，因为它们能够保证状态的改变是安全的。子类推荐被定义为自定义同步组件的静态内部类，同步器自身没有实现任何同步接口，它仅仅是定义了若干同步状态获取和释放的方法来供自定义同步组件使用，同步器既可以支持独占式地获取同步状态，也可以支持共享式地获取同步状态，这样就可以方便实现不同类型的同步组件(ReentrantLock、ReentrantReadWriteLock 和 CountDownLatch 等)。

锁是面向使用者而言的，它一定了使用者与锁交互的借口，比如可以允许两个线程并行访问，隐藏了实现的细节。同步器面向的是锁的实现者，它简化了锁的实现方式，屏蔽了同步状态管理、线程的排队、等待与唤醒等底层操作。总的来说，锁和同步器很好地隔离了使用者和实现者所需关注的领域。

队列同步器的设计是基于模板方法模式的，也就是说，使用者需要继承队列同步器并重写制定的方法，随后将队列同步器组合在自定义同步组件的实现中，并调用同步器提供的模板方法，而这些模板方法将会调用使用者重写的方法。重写同步器指定的方法时，需要使用同步器提供的 3 个方法，getState()、setState(int newState)和 compareAndSetState(int expect,int update)来访问或者修改同步状态。

队列同步器可重写的方法有以下这些：

- (1) tryAcquire：独占式获取同步状态，实现该方法需要查询当前状态并判断同步状态是否符合预期，然后再进行 CAS 设置同步状态。
- (2) tryRelease：独占式释放同步状态，等待获取同步状态的线程有机会获取同步状态。
- (3) tryAcquireShared：共享式获取同步状态，返回大于等于 0 的值，表示获取成功，反之获取失败。
- (4) tryReleaseShared：共享式释放同步状态。
- (5) 当前同步器是否在独占模式下被线程占用，一般该方法表示是否被当前线程所独占。

实现自定义同步组件时，将会调用同步器提供的模板方法，这些模板方法包括以下这些：

- (1) **acquire**：独占式获取同步状态，如果当前线程获取同步状态成功，则由该方法返回，否则，将会进入同步队列等待，该方法将会调用重写的 **tryAcquire** 方法。
- (2) **acquireInterruptibly**：与 **acquire** 方法相同，但是该方法响应中断，当前线程未获取到同步状态而进入同步队列中，如果当前线程被中断，则该方法会抛出 **InterruptedException** 并返回。
- (3) **AcquireNanos**：在 **acquireInterruptibly** 基础上增加了超时限制，如果当前线程在超时时间内没有获取到同步状态，那么将会返回 **false**，如果获取到了返回 **true**。
- (4) **acquireShared**：共享式的获取同步状态，如果当前线程未获取到同步状态，将会进入同步队列等待，与独占式获取的主要区别是在同一时刻可以有多个线程获取到同步状态。
- (5) **acquireSharedInterruptibly**：与 **acquireShared** 相同，该方法响应中断。
- (6) **tryAcquireSharedNanos**：在 **acquireSharedInterruptibly** 基础上增加了超时限制。
- (7) **release**：独占式的释放同步状态，该方法会在释放同步状态之后，将同步队列中第一个节点包含的线程唤醒。
- (8) **releaseShared**：共享式的释放同步状态。
- (9) **getQueuedThreads()**：获取等待在同步队列上的线程集合。

队列同步器提供的模板方法基本上分为 3 类：独占式获取与释放同步状态、共享式获取与释放同步状态，以及查询同步队列中的等待线程情况。自定义同步组件将使用队列同步器提供的模板方法来实现自己的同步语义。

独占锁指的是在同一时刻只能有一个线程获取到锁，而其他获取锁的线程只能在同步队列中等待，只有获取锁的线程释放了锁，后续的线程才能够获取锁。

代码清单 5-32 独占锁示例

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.AbstractQueuedSynchronizer;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;

public class MutexDemo implements Lock{
    //静态内部类，用于自定义同步器
    private static class SyncDemo extends AbstractQueuedSynchronizer{
        //是否处于占用状态
        protected boolean isHeldExclusively(){
            return getState() == 1;
        }
        //当状态为 0 时获取锁
        public boolean tryAcquire(int acquires){
            if(compareAndSetState(0,1)){
```

```
        setExclusiveOwnerThread(Thread.currentThread());
        return true;
    }
    return false;
}
//释放锁, 将状态设置为 0
protected boolean tryRelease(int releases){
    if(getState() == 0){
        throw new IllegalMonitorStateException();
    }
    setExclusiveOwnerThread(null);
    setState(0);
    return true;
}
}
@Override
public void lock() {
    // TODO Auto-generated method stub
}

@Override
public void lockInterruptibly() throws InterruptedException {
    // TODO Auto-generated method stub
}

@Override
public Condition newCondition() {
    // TODO Auto-generated method stub
    return null;
}

@Override
public boolean tryLock() {
    // TODO Auto-generated method stub
    return false;
}

@Override
public boolean tryLock(long arg0, TimeUnit arg1)
    throws InterruptedException {
    // TODO Auto-generated method stub
    return false;
}

@Override
public void unlock() {
```



```
// TODO Auto-generated method stub
```

```
}
```

```
}
```

上面代码 5-32 中，独占锁 `MutexDemo` 是一个自定义同步组件，它在同一时刻只允许一个线程占有锁。`MutexDemo` 中定义了一个静态内部类，该内部类继承了队列同步器并上线了独占式获取和释放同步状态。用户使用独占锁时并不会直接和内部队列同步器的实现打交道，而是调用实现的方法，以获取锁的 `lock()` 方法为例，只需要在方法实现中调用同步器的模板方法 `acquire(int args)` 即可，当前线程调用该方法获取同步状态失败后会被加入到同步队列中等待，这样就大大降低了实现一个可靠自定义同步组件的门槛。

队列同步器依赖内部的一个 FIFO 双向队列来完成同步状态的管理，当前线程获取同步状态失败时，同步器会将当前线程以及等待状态等信息构造成为一个节点并将其加入同步队列，同时会阻塞当前线程，当同步状态释放时，会把首节点中的线程唤醒，使其再次尝试获取同步状态。没有成功获取同步状态的线程将会成为节点加入该队列的尾部。这里我们所说的节点，用来保存获取同步状态失败的线程引用、等待状态以及前驱和后继结点信息。

共享式获取与独占式获取最主要的区别在于同一时刻是否有多个线程同时获取到同步状态。以文件的读写为例，如果一个程序在对文件进行读操作，那么这一时刻对于该文件的写操作均被阻塞，而读操作能够同时进行。写操作要求对资源的独占式访问，而读操作可以是共享式访问。

5.2.5 可重入锁

重入锁就是支持重进入的锁，它表示该锁能够支持一个线程对资源的重复加锁，还支持获取锁时的公平和非公平性选择。Java5 提供了重入锁 `ReentrantLock`² 的实现。

“可重入锁”的概念我们可以理解为，自己可以再次获取自己的内部锁。比如有 1 个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想获取这个对象的锁时还是可以获取的，如果不可锁重入的话，就会造成死锁。可重入锁也支持在父子类继承的环境中工作。

为了实现重进入，我们需要解决以下两个问题：

- (1) 线程再次获得锁。锁需要去识别获取锁的线程是否为当前占据锁的线程，如果是，则再次获取锁。
- (2) 锁的最终释放。线程重复 n 次获取了锁，随后在第 n 次释放该锁时，其他线程能够获取到该锁。锁的最终释放要求锁对于获取进行计数自增，计数表示当前锁被重复获取的次数，而锁被释放时，计数自减，当计数等于 0 时表示锁已经成功释放。

我们之前讲的例子，`MutexDemo`，当一个线程调用 `MutexDemo` 的 `lock()` 方法获取锁之后，如果再次调用 `lock()` 方法，则该线程将会被自己所阻塞，原因是 `MutexDemo` 在实现 `tryAcquire` 方法时没有考虑占有锁的线程再次获取锁的场景，而在调用 `tryAcquire` 方法时返回了 `false`，导致该线

² 同时也是互斥锁，即一次最多只能有一个线程持有的锁，在 jdk1.5 之前，我们通常使用 `synchronized` 机制控制多个线程对共享资源的访问。

程被阻塞。简单地说, `MutexDemo` 是一个不支持重进入的锁。`Synchronized` 关键字隐式地实现了重进入, 比如一个 `synchronized` 修饰的递归方法, 在方法执行时, 执行线程在获取了锁之后仍能连续多次地获得该锁, 而不像独占锁方式出现阻塞自己的情况。

锁获取的公平性问题是, 如果在绝对事件上, 先对锁进行获取的请求一定先被满足, 那么这个锁是公平的, 反之, 是不公平的。公平的获取锁, 也就是等待时间最长的线程最优先获取锁, 也可以说获取锁是一个顺序的行为。`ReentrantLock` 提供了一个构造函数, 能够控制锁是否是公平的。事实上, 公平的锁机制没有不公平的效率, 但是, 并不是任何场景都是以 TPS 作为唯一的指标, 公平锁能够减少低优先级线程等待发生的概率, 等待越久的请求越是能够得到优先满足。

`ReentrantLock` 虽然没能像 `Synchronized` 关键字一样支持隐式的重进入, 但是在调用 `lock()` 方法时, 已经获取到锁的线程, 能够再次调用 `lock()` 方法获取锁而不被阻塞。

`ReentrantLock` 拥有 `Synchronized` 相同的并发性和内存语义, 此外还多了锁投票、定时锁等候和中断锁等候等机制。

线程 A 和 B 都要获取对象 O 的锁定, 假设 A 获取了对象 O 锁, B 将等待 A 释放对 O 的锁定, 如果使用 `synchronized`, 如果 A 不释放, B 将一直等下去, 不能被中断。如果使用 `ReentrantLock`, 如果 A 不释放, 可以使 B 在等待了足够长的时间以后, 中断等待, 而干别的事情。

`ReentrantLock` 获取锁定四种方式。

- (1) `lock()`, 如果获取了锁立即返回, 如果别的线程持有锁, 当前线程则一直处于休眠状态, 直到获取锁。
- (2) `tryLock()`, 如果获取了锁立即返回 `true`, 如果别的线程正持有锁, 立即返回 `false`。
- (3) `tryLock(long timeout, TimeUnit, unit)`, 如果获取了锁定立即返回 `true`, 如果别的线程正持有锁, 会等待参数给定的时间, 在等待的过程中, 如果获取了锁定, 就返回 `true`, 如果等待超时, 返回 `false`;
- (4) `lockInterruptibly`, 如果获取了锁定立即返回, 如果没有获取锁定, 当前线程处于休眠状态, 直到锁定或者当前线程被别的线程中断。

`synchronized` 是在 JVM 层面上实现的, 不但可以通过一些监控工具监控 `synchronized` 的锁定, 而且在代码执行时出现异常, JVM 会自动释放锁定, 但是使用 `Lock` 则不行, `lock` 是通过代码实现的, 要保证锁定一定会被释放, 就必须将 `unlock()` 放到 `finally{}` 中。

在资源竞争不是很激烈的情况下, `Synchronized` 的性能要优于 `ReentrantLock`, 但是在资源竞争很激烈的情况下, `Synchronized` 的性能会下降几十倍, 但是 `ReentrantLock` 的性能可以维持常态。

5.2.6 读写锁

独占锁和可重入锁都属于排他锁, 这些锁在同一时刻只允许一个线程进行访问, 而读写锁在同一时刻可以允许多个读线程访问, 但是在写线程访问时, 所有的读线程和其他写线程均被阻塞。读写锁维护了一对锁, 一个读锁和一个写锁, 通过分离读锁和写锁, 使得并发性相比一般的排他锁有了很大提升。

Java 提供的读写锁是 `ReentrantReadWriteLock`, 注意 `ReentrantReadWriteLock` 和 `ReentrantLock`

不是继承关系，但都是基于 `AbstractQueuedSynchronizer` 来实现。

在没有读写锁支持之前，如果需要完成上述工作就要使用 Java 的等待通知机制，就是当写操作开始时，锁有晚于写操作的读操作均会进入等待状态，只有写操作完成并进行通知之后，所有等待的读操作才能继续执行，注意，写操作之间依靠 `Synchronized` 关键字进行同步。这样做的目的是使读操作能够读取到正确的数据，不会出现脏读。读写锁也能够简化读写交互场景的编程方式。例如，程序中定义一个共享的用作缓存数据结构，它大部分时间提供读服务，包括查询和检索，而写操作占有的时间很少，写操作完成之后的更新需要对后续的服务可见。

改用读写锁实现上述功能，只需要在读操作时获取读锁，写操作时获取写锁即可。当写锁被获取到时，后续（非当前写操作线程）的读写操作都会被阻塞，写锁释放之后，锁有操作继续执行，编程方式相对于使用等待通知机制的实现方式而言，变得简单明了。注意，在同一线程中，持有读锁后，不能直接调用写锁的 `lock` 方法，否则会造成死锁。

一般情况下，读写锁的性能都会比排他锁好，因为大多数场景读都比写多，读写锁能够提供比排他锁更好的并发性和吞吐量。我们可以把读写锁的特性总结为以下几个。

- 读取锁允许多个 `reader` 线程同时持有，而写入锁最多只能有一个 `writer` 线程持有。
- 读写锁的使用场合：读取共享数据的频率远大于修改共享数据的频率。在上述场合下，使用读写锁控制共享资源的访问，可以提高并发性能。
- 如果一个线程已经持有了写入锁，则可以再持有读写锁。相反，如果一个线程已经持有了读取锁，则在释放该读取锁之前不能再持有写入锁。
- 可以调用写入锁的 `newCondition()` 方法获取与该写入锁绑定的 `Condition` 对，此时与普通的互斥锁并没有什么区别。但是调用读取锁的 `newCondition()` 方法将抛出异常。

5.2.7 偏向锁和轻量级锁

Java6 为了减少获得锁和释放锁带来的性能消耗，引入了“偏向锁”和“轻量级锁”，在 Java6 种，锁一共有 4 种状态，级别从低到高依次是：无锁状态、偏向锁状态、轻量级锁状态和重量级锁状态，这几个状态会随着竞争情况逐渐升级。锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁。这种锁升级却不能降级的策略，目的是为了提高获得锁和释放锁的效率。

1. 偏向锁

大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低而引入了偏向锁。当一个线程访问同步块获取锁时，会在对象头和栈帧中的锁记录里存储锁偏向的线程 ID，以后该线程在进入和退出同步块时不需要进行 CAS 操作来加锁和解锁，只需简单地测试一下对象头的 `Mark Word` 里是否存储着指向当前线程的偏向锁。如果测试成功，表示线程已经获得了锁。如果测试失败，则需要再测试一下 `Mark Word` 中偏向锁的标识是否设置成了 1（表示当前是偏向锁）：如果没有设置，则使用 CAS 竞争锁；如果设置了，则尝试使用 CAS 将对象头的偏向锁指向当前线程。

1) 偏向锁的撤销

偏向锁使用了一种等到竞争出现才释放锁的机制，所以当其他线程尝试竞争偏向锁时，持有

偏向锁的线程才会释放锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有正在执行的字节码）。它会首先暂停拥有偏向锁的线程，然后检查持有偏向锁的线程是否活着，如果线程不处于活动状态，则将对象头设置成无锁状态；如果线程仍然活着，拥有偏向锁的栈会被执行，遍历偏向对象的锁记录，栈中的锁记录 and 对象头的 Mark Word 要么重新偏向于其他线程，要么恢复到无锁或者标记对象不适合作为偏向锁，最后唤醒暂停的线程。

2) 关闭偏向锁

偏向锁在 Java6 和 Java7 里是默认启用的，但是它在应用程序启动几秒钟之后才被激活，如有必要可以使用 JVM 参数来关闭延迟：-XX:BiasedLockingStartupDelay=0。如果你确定应用程序里所有的锁通常情况下处于竞争状态，可以通过 JVM 参数关闭偏向锁：-XX:UseBiasedLocking=false，那么程序默认会进入轻量级锁状态。

2. 轻量级锁

1) 轻量级锁加锁

线程在执行同步块之前，JVM 会先在当前线程的栈帧中创建用于存储锁记录的空间，并将对象头中的 Mark Word 复制到锁记录中，官方称为 Displaced Mark Word。然后线程尝试使用 CAS 将对象头中的 Mark Word 替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。

2) 轻量级锁解锁

轻量级解锁时，会使用原子的 CAS 操作将 Displaced Mark Word 替换回到对象头，如果成功，则表示没有竞争发生。如果失败，表示当前锁存在竞争，锁就会膨胀成重量级锁。

因为自旋会消耗 CPU，为了避免无用的自旋（比如获得锁的线程被阻塞住了），一旦锁升级成重量级锁，就不会恢复到轻量级状态。当锁处于这个状态下，其他线程试图获取锁时，都会被阻塞住，当持有锁的线程释放锁之后会唤醒这些线程，被唤醒的线程就会进行新一轮的夺锁之争。

锁的优缺点的对比如表 5-1 所示。

表 5-1 锁对比表

锁	优点	缺点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非亦同步方法相比仅存在纳秒级的差距	如果线程间存在锁竞争，会带来额外的锁撤销的消耗	适用于只有一个线程访问同步块场景
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度	如果始终得不到竞争的线程，使用自旋会消耗 CPU	追求响应时间，同步快执行速度非常快
重量级锁	线程竞争不使用自旋，不会消耗 CPU	线程阻塞，响应时间缓慢	追求吞吐量，同步块执行速度较长

5.3 增加程序并行性

现代 CPU 架构将多核、多硬件执行线程技术摆到了程序员面前。这意味着我们可以利用更多

的 CPU 资源做更多的工作。然而，要利用好这些额外的 CPU 资源，运行于其上的程序必须要能够并行工作。换句话说，这些程序需要按照多线程的方式构造或设计才能充分利用额外的硬件线程。

单线程的 Java 应用程序无法充分利用现代 CPU 架构上额外的硬件线程。那些单线程应用必须按照多线程的方式重构才能并行工作。此外，很多 Java 应用程序都有单线程阶段或操作，尤其是初始化或启动阶段。通过并行执行任务、同事利用多个线程，这些 Java 应用程序的初始化或启动性能将大幅提升。

5.3.1 并发计数器

多核时代来临了之后，大家都开始使用并发计数器，我们先来看一下 Java 迄今为止提供了哪些实现方式，它们的性能和这个新的 API 相比，又有什么不同。

脏计数器，选择这种方式意味着多线程直接并发读写一个普通对象或者静态字段。不幸的是，这么做是行不通的。有两个原因，一是在 Java 里， $A+=B$ 操作不是原子的。如果你打开编译后的字节码看一下，你会发现至少有四条指令，第一条是从堆里将字段值加载到线程栈里，第二条是加载要增加的值，第三条指令将它们进行相加，第四条则将结果写回到字段中。如果多个线程同时在同一个内存位置进行这个操作，你的一个写操作很有可能就丢掉了，因为另一个线程可能会覆盖了它的值。还有一个很恶心的事就是这个值的可见性。

synchronized 关键字，它是最基础的同步操作了，只要你在读写值，它就会阻塞住其他的所有线程。这种方式的确行得通，不过有一点事可以肯定的，你的程序会排长队的。

读写锁 (RWLock)，这个和基础的 Java 锁相比就巧妙了些，它可以让你区分出那些要修改值因此需要阻塞别人的进程以及那些只是读取值不需要进入临界区的。虽然这个方法有的时候很高效（比如写线程的数量比较少的话），但还是相当无语，因为当你获取写锁的时候还是会阻塞住其他线程的执行。

volatile 关键字，这个经常会被误用的关键字会让 JIT 编译停止在运行时进行机器码的优化工作，因此字段一旦有更新别的线程马上就能看到。它会使得 JIT 编译器经常玩的一些把戏比如说调整赋值语句的顺序这些无法进行。JIT 编译器有可能会改变字段的赋值顺序。什么，你再说一遍？是的，你听的没错。这个神秘的小把戏使得它可以减小程序访问全局堆的次数，同时它还能保证不会影响到你的程序的执行。这真是有点偷偷摸摸的感觉。

那什么时候应该使用 volatile 计数器？如果你只有一个线程在更新一个值，而多个线程在读的话，这是个很合适的场景。因为完全没有竞争。

你可能会问为什么不都使用它就完了？因为如果有多个线程在更新的话就会有问题了。由于 $A+=B$ 不是一个原子操作，这么做的话可能会覆盖掉别人写的话。在 Java 8 以前，这种情况你就只能用 AtomicInteger 了。

AtomicInteger，这组类使用了处理器的 CAS (compare-and-swap) 指令来更新计数器的值。听起来不错吧？一半一半吧。由于它直接使用机器指令来设置值，因此对其他线程的影响最小。不好的一面是如果它和别的线程有竞争赋值失败了，它会继续重试。在高并发的条件下，这就成了一个自旋锁，线程会在一个无限的循环内不断的尝试赋值，直到成功为止。我们可不太想看到这

种局面。Java 8 来了，还带来了 LongAdders。

Adders，这是 Java8 非常棒的新的 API，从使用者的角度来说，它很像 AtomicInteger。只需要创建一个 LongAdder 对象，然后使用 intValue()以及 add()方法来获取和设置它的值。而奇迹就发生在这一切的背后。如果由于竞争这个类的 CAS 操作失败的话，它会要添加的值存到一个线程本地的内部的 cell 对象里。当 intValue()方法调用的时候，它把这些 cell 的值加到总和里，就减少了 CAS 重试或者阻塞别的线程的情况。

假设我们来做一个基准测试，把一个计数器设置为 0，然后多个线程开始读取并进行自增。当计数器到达 10^8 的时候停止。假设我们在一个 4 核的 i7 处理器上运行这个测试。

我用了 10 个线程来运行这个基准测试——读写分别使用 5 个线程来进行，这样的话会出现严重的竞争条件，如图 5-2 所示。注意，脏读和 volatile 都有可能产生脏值。

DIRTY	VOLATILE	SYNCHRONIZED	RWLOCK	ATOMIC	ADDER
1129	4543	12091	133555	2591	1590
1101	4679	13177	132676	3266	1642
1079	4077	17157	180287	3128	1459
1124	3662	16123	184923	3262	1560
1089	4723	16366	179975	3117	1514
1117	4535	16647	155151	3014	1403
1100	3615	17421	138867	2759	1331
1143	3505	17009	147162	3147	1375
317	5005	17173	169682	3081	1409
978	4786	17296	160799	2808	1421
1017.7	4313	16046	158307.7	3017.3	1470.4

图5-2 读写基准测试

总的来说，并发的 Adder 类和 AtomicInteger 相比有 60~100%的性能提升。此外，增加线程不会对结果有太大影响，除非是使用锁的情况。最后我们需要注意到，如果使用 synchronized 或者读写锁，性能会有很大的损耗，慢至少一个数量级。

5.3.2 减少上下文切换次数

多任务系统往往需要同时执行多道作业。如果作业数大于机器的 CPU 数，一颗 CPU 同时只能执行一项任务，那么为了让用户感觉这些任务正在同时进行，操作系统的设计者巧妙地利用了时间片轮转的方式，CPU 给每个任务都服务一定的时间，然后把当前任务的状态保存下来，在加载下一任务的状态后，继续服务下一任务。任务的状态保存及再加载，这段过程就叫作上下文切换。时间片轮转的方式使多个任务在同一颗 CPU 上执行变成了可能，但同时也带来了保存现场和加载现场的直接消耗。更精确的说法是，上下文切换会带来直接和间接两种因素影响程序性能的消耗。这些直接消耗包括 CPU 寄存器需要保存和加载、系统调度器的代码需要执行、TLB³实例需

³ TLB(Translation Lookaside Buffer)传输后备缓冲器是一个内存管理单元用于改进虚拟地址到物理地址转换速度的缓存。

要重新加载、CPU 的 pipeline⁴需要被清空、间接消耗指的是多核的 cache 之间的共享数据、间接消耗对于程序的影响要看线程工作区操作数据的大小等。

上下文切换又分为 2 种：让步式上下文切换和抢占式上下文切换。前者是指执行线程主动释放 CPU，与锁竞争严重程度成正比，可通过减少锁竞争来避免；后者是指线程因分配的时间片用尽而被迫放弃 CPU 或者被其他优先级更高的线程所抢占，一般由于线程数大于 CPU 可用核心数引起，可通过调整线程数，适当减少线程数来避免。

现在 Linux 是大多基于抢占式，CPU 给每个任务一定的服务时间，当时间片轮转的时候，需要把当前状态保存下来，同时加载下一个任务，这个过程叫作上下文切换。时间片轮转的方式，使得多个任务利用一个 CPU 执行成为可能，但是保存现场和加载现场，也带来了性能消耗。我们可以通过 VMSTAT 命令来获取上下文切换的次数。

代码清单 5-33 VMSTAT 获取上下文切换的次数

```
[root@zuoyel ~]# vmstat 1 100
procs -----memory----- --swap-- -----io----- --system-- -----cpu-----
 r b  swpd   free   buff  cache   si   so    bi   bo    in  cs us sy id wa st
 1  0        0 2466584 172864 3091060    0    0     5   94   35 157  2  5 91  2  0
```

Vmstat 命令返回的结果是系统层面的，如果想看查看特定进程的情况，可以使用 pidstat。

代码清单 5-34 pidstat 返回进程上下文切换情况

```
[root@zuoyel ~]# pidstat
Linux 2.6.32-504.el6.x86_64 (zuoyel)    11/24/2015    _x86_64_    (4 CPU)
10:23:33 AM      PID    %usr %system %guest    %CPU   CPU Command
10:23:33 AM        1     0.00   0.00   0.00    0.00     3  init
10:23:33 AM        3     0.00   0.22   0.00    0.22     0 migration/0
10:23:33 AM        4     0.00   0.01   0.00    0.01     0 ksoftirqd/0
10:23:33 AM        6     0.00   0.00   0.00    0.00     0 watchdog/0
10:23:33 AM        7     0.00   0.11   0.00    0.11     1 migration/1
```

- PID: 进程 pid;
- %usr: 进程在用户态运行所占 cpu 时间比率;
- %system: 进程在内核态运行所占 cpu 时间比率;
- %CPU: 进程运行所占 cpu 时间比率;
- CPU: 指示进程在哪个核运行;
- Command: 拉起进程对应的命令。

注意，执行 pidstat 默认输出信息为系统启动后到执行时间点的统计信息，因而即使当前某进程的 cpu 占用率很高，输出中的值有可能仍为 0。

⁴ 流水线是 Intel 首次在 486 芯片中开始使用的。流水线的工作方式就象工业生产上的装配流水线。在 CPU 中由 5 到 6 个不同功能的电路单元组成一条指令处理流水线，然后将一条 X86 指令分成 5 到 6 步后再由这些电路单元分别执行，这样就能实现在一个 CPU 时钟周期完成一条指令，因此提高 CPU 的运算速度。

Context Switch 过高，会导致 CPU 像个搬运工，频繁在寄存器和运行队列直接奔波，更多的时间花在了线程切换，而不是真正工作的线程上。直接的消耗包括 CPU 寄存器需要保存和加载，系统调度器的代码需要执行。间接消耗在于多核 cache 之间的共享数据。

综合上面的内容，我们可以知道真正干活的不是线程，而是 CPU。线程越多，干活不一定越快，我们需要减少上下文切换次数，会让系统更快速。

那么高并发的情况下什么时候适合单线程，什么时候适合多线程呢？

适合单线程的场景：单个线程的工作逻辑简单，而且速度非常快，比如从内存中读取某个值，或者从 Hash 表根据 key 获得某个 value。Redis 和 Node.js 这类程序都是单线程，适合单个线程简单快速的场景。

适合多线程的场景：单个线程的工作逻辑复杂，等待时间较长或者需要消耗大量系统运算资源，比如需要从多个远程服务获得数据并计算，或者图像处理。

5.3.3 针对 Thread 类的更新

Java7 对于表示线程的类 `java.lang.Thread` 所做的更新主要明确了 `Thread` 类的对象在某些情况下的行为，并且去掉了之前使用中比较模糊的和设计不合理的部分。首先将 `Thread` 类的 `clone` 方法改进为总是抛出 `CloneNotSupportedException` 异常，这是因为对一个 `Thread` 类的对象进行克隆是没有意义的。Java7 显式地禁止了对 `Thread` 类对象的克隆操作。其次，在 Java7 之前，`Thread` 类的 `join` 方法和 `sleep` 方法可以接收一个 `long` 类型的参数表示等待的时间，但是并没有定义当这个参数为负数时的处理方式。Java7 中规定：如果这两个方法的等待时间参数的值为负数，则会抛出 `IllegalArgumentException` 异常。

另外一个明确了参数处理行为的是 `Thread` 类的构造方法。在创建 `Thread` 类的对象时可以使用参数包括：表示 `Thread` 类的对象所在线程组的 `java.lang.ThreadGroup` 类的对象，表示需要运行的任务的 `java.lang.Runnable` 接口的实现对象，以及表示线程名称的 `String` 类的对象。如果传入的 `ThreadGroup` 类的对象为 `null`，那么会先尝试调用当前配置好的安全管理器（`java.lang.SecurityManager` 类的对象）的 `getThreadGroup` 方法来获取 `ThreadGroup` 类的对象；如果没有配置安全管理器或 `getThreadGroup` 方法也返回 `null`，那么会使用当前线程所在线程组的 `ThreadGroup` 类的对象；如果传入的 `Runnable` 接口的实现对象为 `null`，那么会调用 `Thread` 类的对象本身的 `run` 方法；如果传入的线程名称是 `null`，会抛出 `NullPointerException` 异常。

在调用 `Thread` 类的 `setContextClassLoader` 方法来设置线程上下文类加载器时，如果传入的参数为 `null`，则表明使用的是系统类加载器。如果无法使用系统类加载器，就使用启动类加载器。同样的，如果当前线程上下文加载器是系统类加载器或启动类加载器，那么 `getContextClassLoader` 方法的返回值是 `null`。

5.3.4 Fork/Join 框架

在 Java5 版本的时候，Java 设计者们通过 JSR-166 的规范制定，在 `java.util.concurrent` 包下为开发人员提供了基于粗粒度的多核并行计算框架，只不过这种基于粗粒度的并行计算模式，并不能在处理效率上达到令人满意的程度。因为这种基于粗粒度的并行计算，根本无法高效组合所有可用物理核心一起进行并行任务处理，甚至在某些情况下，还有可能导致部分物理核心处于空闲

的状态。所以 Java7 版本在 `java.util.concurrent.forkjoin` 包下新增了基于细粒度的多核并行计算 Fork/Join 框架。该框架的设计初衷是将一个任务量化到最小，并提供高计算密度的并行处理性能。简单来说，我们可以将一个任务拆分成若干个子任务，直到这个任务足够小，然后每一个子任务被独立并行计算，直到任务执行完成，再将其逐个合并为一个完整的任务，这就是 Fork/Join 框架提供的细粒度并行计算模式。

前面说过，Fork/Join 框架是 Java7 提供的一个用于并行任务执行的框架，是把一个大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。从英文字面上也可以理解，Fork 就是把一个大任务切分为若干子任务并行的执行，Join 就是合并这些子任务的执行结果，最后得到这个大任务的结果。比如计算 $1+2+\dots+10000$ ，可以分割为 100 个任务，每个子任务分别对 100 个数进行求和，最终汇总这 10 个子任务的结果。其原理如图 5-3 所示。

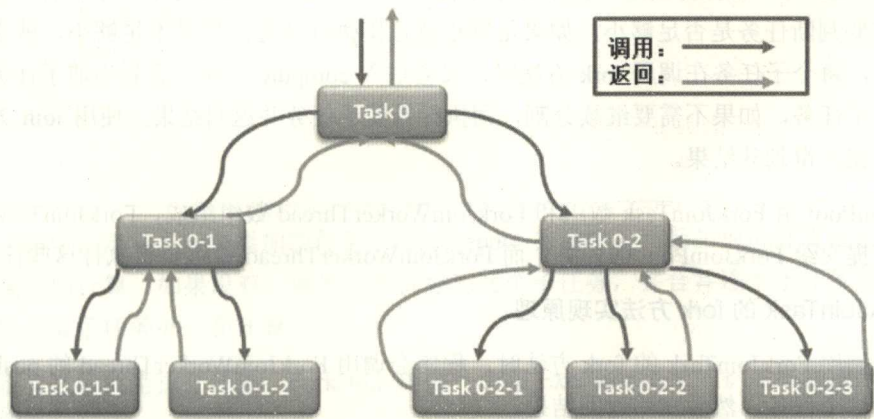


图5-3 Fork/Join原理图

Fork/Join 框架的工作分为两个步骤：

- (1) 分割任务。首先需要有一个 `fork` 类来把大任务分割成子任务，有可能子任务还是很大，所以还需要不停地分割，直到分割出的子任务足够小。
- (2) 执行任务并合并结果。分割的子任务分别放在双端队列里，然后几个启动线程分别从双端队列里获取任务执行。子任务执行完的结果都统一放在一个队列里，启动一个线程从队列里拿数据，然后合并这些数据。

Fork/Join 相关类的类间关系如图 5-4 所示。

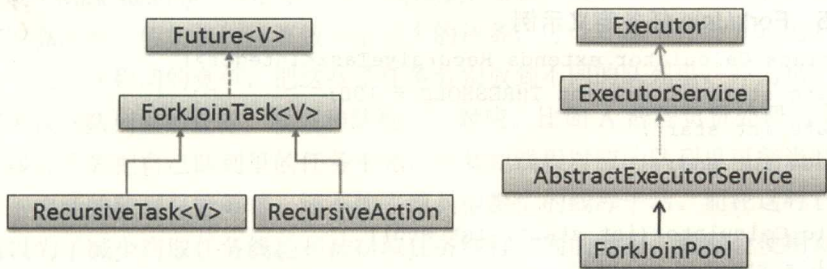


图5-4 Fork/Join类图

Fork/Join 具体使用两个类来完成以上两件事情。

- (1) ForkJoinTask: 我们要使用 ForkJoin 框架, 必须首先创建一个 ForkJoin 任务。它提供在任务中执行 fork() 和 join() 操作的机制。通常情况下, 我们不需要直接继承 ForkJoinTask 类, 只需要继承它的子类, Fork/Join 框架提供了以下两个子类。

RecursiveAction: 用于没有返回结果的任务。

RecursiveTask: 用于有返回结果的任务。

- (2) ForkJoinPool: ForkJoinTask 需要通过 ForkJoinPool 来执行。

任务分割出的子任务会添加到当前工作线程所维护的双端队列中, 进入队列的头部。当一个工作线程的队列里暂时没有任务时, 它会随机从其他工作线程的队列的尾部获取一个任务。

ForkJoinTask 与一般任务的主要区别在于它需要事先调用 compute 方法用于计算, 在这个方法里, 首先需要判断任务是否足够小, 如果足够小就直接执行任务。如果不够小, 就必须分割成两个子任务, 每个子任务在调用 fork 方法时, 又会进入 compute 方法, 看看当前子任务是否需要继续分割成子任务, 如果不需要继续分割, 则执行当前子任务并返回结果。使用 join 方法会等待子任务执行完并得到其结果。

ForkJoinPool 由 ForkJoinTask 数组和 ForkJoinWorkerThread 数组组成, ForkJoinTask 数组负责将存放程序提交给 ForkJoinPool 的任务, 而 ForkJoinWorkerThread 数组负责执行这些任务。

1. ForkJoinTask 的 fork 方法实现原理

当我们调用 ForkJoinTask 的 fork 方法时, 程序会调用 ForkJoinWorkerThread 的 pushTask 方法异步地执行这个任务, 然后立即返回结果。

pushTask 方法把当前任务存放在 ForkJoinTask 数组队列里。然后再调用 ForkJoinPool 的 signalWork() 方法唤醒或创建一个工作线程来执行任务。

2. ForkJoinTask 的 join 方法实现原理

Join 方法的主要作用是阻塞当前线程并等待获取结果。

首先, Join 方法会调用 doJoin() 方法, 通过 doJoin() 方法得到当前任务的状态来判断返回什么结果, 任务状态有 4 种: 已完成 (NORMAL)、被取消 (CANCELLED)、信号 (SIGNAL) 和出现异常 (EXCEPTIONAL)。

Fork/Join 的任务定义如下例所示。

代码清单 5-35 Fork/Join 任务定义示例

```
public class Calculator extends RecursiveTask<Integer>{
    private static final int THRESHOLD = 100;
    private int start;
    private int end;

    public Calculator(int start, int end){
        this.start = start;
        this.end = end;
    }
}
```



```

    }

    @Override
    protected Integer compute() {
        int sum = 0;
        if((start - end) < THRESHOLD){
            for(int i = start; i < end; i++){
                sum += i;
            }
        }else{
            int middle = (start + end) / 2;
            Calculator left = new Calculator(start, middle);
            Calculator right = new Calculator(middle + 1, end);
            left.fork();
            right.fork();
            sum = left.join() + right.join();
        }
        return sum;
    }
}

```

这段代码中，定义了一个累加的任务，在 `compute` 方法中，判断当前的计算范围是否小于一个值，如果是则计算，如果没有，就把任务拆分为连个子任务，并合并连个子任务的中间结果。程序递归地完成了任务拆分和计算。

任务定义之后就是执行任务，Fork/Join 提供一个和 Executor 框架的扩展线程池来执行任务，代码如下所示。

代码清单 5-36 执行任务示例

```

@Test
public void run() throws Exception{
    ForkJoinPool forkJoinPool = new ForkJoinPool();
    Future<Integer> result = forkJoinPool.submit(new Calculator(0, 10000));
    assertEquals(new Integer(49995000), result.get());
}

```

为了确保 Fork/Join 方式可以奏效，JDK 采用工作窃取算法来保证多个线程可以访问多个队列。

工作窃取（work-stealing）算法是指某个线程从其他队列里窃取任务来执行。那么为什么需要使用工作窃取算法呢？假如我们需要做一个较大的任务，可以把这个任务分割为若干互不依赖的子任务，为了减少线程间的竞争，把这些子任务分别放到不同的队列里，并为每个队列创建一个单独的线程来执行队列里的任务，线程和队列一一对应。比如 A 线程负责处理 A 队列里的任务。但是，有的线程会先把自己队列里的任务干完，而其他线程对应的队列里可能当时还有任务等待处理。这样干完活的队列只能等着，还不如去帮其他繁忙的线程干活。而在这时它们会访问同一个队列，所以为了减少窃取任务线程和被窃取任务线程之间的竞争，通常会使用双端队列，被窃取任务线程永远从双端队列的头部拿任务执行，而窃取任务的恶线程永远从双端队列的尾部拿任

务执行。工作窃取的流程图如图 5-5 所示。

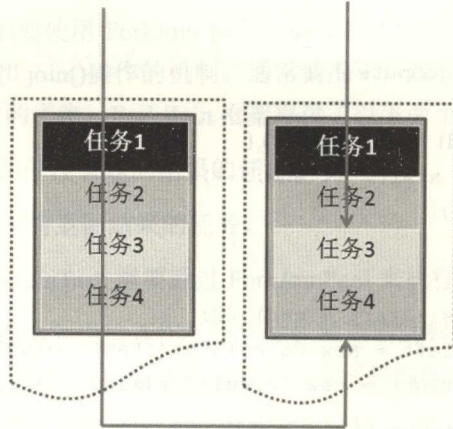


图5-5 工作窃取流程图

工作窃取算法的优点是它能够充分利用线程进行并行计算，减少了线程间的竞争。缺点是在某些情况下还是存在竞争，比如双端队列里只有一个任务时。并且该算法会消耗更多的系统资源，比如创建多个线程和多个双端队列。

除了 Java7 能够高效地利用 Fork/Join 框架实现多核并行计算外，开源基金会 Apache 提供的 Hadoop MapReduce 框架也是一个高效的海量数据计算框架，它允许开发人员将其部署在廉价的集群服务器上处理 TB 级别的数据。当然还有一些编程语言自诞生那天起，就是为了解决并行计算而来，比如 Scala、Clojure 和 Erlang 等。这类编程语言，不仅继承了面向对象的特性，同时还结合了函数式编程等特性。虽然目前 Java 同样也可能使用函数式编程，但代码将会显得非常冗余，不过 Java 设计者们在 Java8 中提供对 Lambda 的支持，这极大地改善了 Java 对于函数式编程的不足。

5.3.5 Executor 框架

在 Java 中，使用线程来异步执行任务。Java 线程的创建与销毁需要一定的开销，如果我们为每一个任务创建一个新线程来执行，这些线程的创建与销毁将消耗大量的计算资源。同时，为每一个任务创建一个新线程来执行，这种策略可能会使处于高负荷状态的应用最终崩溃。

Java 的线程既是工作单元，也是执行机制。从 JDK5 开始，把工作单元与执行机制分离开来。工作单元包括 Runnable 和 Callable，而执行机制由 Executor 框架提供。

在 HotSpot VM 的线程模型中，Java 线程被一对一映射为本地操作系统线程。Java 线程启动时会创建一个本地操作系统线程；当该 Java 线程被终止时，这个操作系统线程也会被回收。操作系统会调度所有线程并将它们分配给可用的 CPU。在上层，Java 多线程程序通常把应用分解为若干个任务，然后使用用户级的调度器（Executor 框架）将这些任务映射为固定数量的线程；在底层，操作系统内核将这些线程映射到硬件处理器上。

Executor 框架主要由 3 大部分组成。

- 任务：包括被执行任务需要实现的接口，Runnable 接口或 Callable 接口。
- 任务的执行：包括任务执行机制的核心接口 Executor，以及继承自 Executor 的

ExecutorService 接口。Executor 框架有两个关键类实现了 ExecutorService 接口 (ThreadPoolExecutor 和 ScheduledThreadPoolExecutor)。

- 异步计算的结果：包括接口 Future 和实现 Future 接口的 FutureTask 类。

Executor 是一个接口，它是 Executor 框架的基础，它将任务的提交与任务的执行分离开来。

ThreadPoolExecutor 是线程池的核心实现类，用来执行被提交的任务。

ScheduledThreadPoolExecutor 是一个实现类，可以在给定的延迟后运行命令，或者定期执行命令。ScheduledThreadPoolExecutor 比 Timer 更灵活，功能更强大。

Future 接口和实现 Future 接口的 FutureTask 类，代表异步计算的结果。

Runnable 接口和 Callable 接口的实现类，都可以被 ThreadPoolExecutor 或 ScheduledThreadPoolExecutor 执行。

主线程首先把要创建实现 Runnable 或者 Callable 接口的任务对象。工具类 Executors 可以把一个 Runnable 对象封装为一个 Callable 对象。然后可以把 Runnable 对象直接交给 ExecutorService 执行，或者也可以把 Runnable 对象或 Callable 对象提交给 ExecutorService 执行。如果执行 ExecutorService.submit(), ExecutorService 将返回一个实现 Future 接口的对象。由于 FutureTask 实现了 Runnable，程序员也可以创建 FutureTask，然后直接交给 ExecutorService 执行。最后，主线程可以执行 FutureTask.get()方法来等待任务执行完成。主线程也可以执行 FutureTask.cancel 来取消任务的执行。

ThreadPoolExecutor：通常使用工厂类 Executors 创建。Executors 可以创建 3 种类型的 ThreadPoolExecutor，即 SingleThreadExecutor、FixedThreadPool 和 CachedThreadPool。

- (1) FixedThreadPool，创建使用固定线程数的 Executors。适用于为了满足资源管理的需求，而需要限制当前线程数量的应用场景，它适用于负载比较重的服务器。
- (2) SingleThreadExecutor，用于创建单个线程的 SingleThreadExecutor，适用于需要保证顺序执行各个任务，并且在任意时间点，不会有多个线程是活动的应用场景。
- (3) CachedThreadPool，创建一个会根据需要创建新线程的 CachedThreadPool，它是大小无界的线程池，适用于执行很多的短期异步任务的小程序，或者是负载较轻的服务器。

5.4 JDK 类库使用

并行程序开发经常提到线程安全，什么叫线程安全？这个首先要明确。线程安全就是说多线程访问同一代码，不会产生不确定的结果。

其次我们需要明确并行和并发区别，如下所示。

- (1) 并行是指两者同时执行一件事，比如赛跑，两个人都在不停地往前跑。
- (2) 并发是指资源有限的情况下，两者交替轮流使用资源，比如一段路(单核 CPU 资源)同时只能过一个人，A 走一段后，让给 B，B 用完继续给 A，交替使用，目的是提高效率。

5.4.1 原子值

原子操作是不可被中断的一个或一系列操作。在多处理器上实现原子操作就变得有点复杂。从 Java5 开始, `java.util.concurrent.atomic` 包提供了用于支持无锁可变变量的类。例如, `AtomicLong` 是作用是对长整形进行原子操作。在 32 位操作系统中, 64 位的 `long` 和 `double` 变量由于会被 JVM 当作两个分离的 32 位来进行操作, 所以不具有原子性。而使用 `AtomicLong` 能让 `long` 的操作保持原子型。

例如, 你可以用如下代码完全地生成一组数字。

代码清单 5-37 生成一组数字

```
public static AtomicLong nextNumber = new AtomicLong();
//在某些线程中。。。
long id = nextNumber.incrementAndGet();
```

`incrementAndGet` 方法会自动将 `AtomicLong` 的值加 1, 并返回增加后的值。即该操作会获得它的值, 增加 1, 再将增加后的值设置给它, 并且产生新值的过程不能被打断。它可以保证即便有多个线程同时并发访问同一个实例, 也能够计算并返回正确的值。

Java5 中提供了很多设置、增加、减少值得原子操作, 但是如果想要进行更复杂的更新操作, 就必须使用 `compareAndSet` 方法。例如, 假设你想要追踪由不同线程所检测的最大值, 那么下面的代码就不行了。

代码清单 5-38 错误的代码

```
public static AtomicLong largest = new AtomicLong();
//在某些线程中...
largest.set(Math.max(largest.get(), observed)); //错误-竞争条件!
```

这个更新过程不是原子性。相反, 应该在一个循环中使用 `compareAndSet` 来计算新值。

代码清单 5-39 使用 `compareAndSet` 方法

```
do{
    oldValue = largest.get();
    newValue = Math.max(oldValue, observed);
}while(!largest.compareAndSet(oldValue, newValue));
```

如果另一个线程也在更新 `largest`, 很可能它已经捷足先登更新成功了。那么随后 `compareAndSet` 会返回 `false`, 并且不会设置新值。此时, 程序会再次尝试循环, 读取更新后的值并试图改变它。最终, 它成功地将已有值替换为了新的值。`compareAndSet` 方法会映射为一个底层的处理器方法, 这远比使用一个锁要快得多。

在 Java8 中不必再编写循环逻辑。只需要提供一个用来更新值得 `lambda` 表达式, 更新操作就会自动完成。在我们的操作中, 可以调用。

代码清单 5-40 Java8 方式

```
largest.updateAndGet(x->Math.max(x, observed));
或者
```

```
largest.accumulateAndGet(observed, Math::max);
```

AccumulateAndGet 方法通过一个二元运算符将原子值传入的参数组合起来。

除了它之外, Java8 还提供了返回原始值得 getAndUpdate 方法和 getAndAccumulate 方法。

注意: AtomicInteger、AtomicIntegerArray、AtomicIntegerFieldUpdater、AtomicLongArray、AtomicLongFieldUpdater、AtomicReference、AtomicReferenceArray 和 AtomicReferenceFieldUpdater 类都提供了这些方法。

当你有大量线程访问同一个原子值时, 由于乐观锁更新需要太多次重试, 因此会导致性能严重下降。为此, Java8 提供了 LongAdder 和 LongAccumulator 来解决该问题。LongAdder 由多个变量组成, 这些变量累加的值即为当前值。多个线程可以更新不同的被加数, 当线程数量增加时自动增加新的被加数。由于通常情况下都是直到所有工作完成后才需要总和值, 所以这种方法效率很高, 它所带来的性能提升十分可观。

如果你的环境中存在高度竞争, 那么就应当用 LongAdder 来代替 AtomicLong。二者之间的方法命名稍有不同。Increment 方法用来将计数器自增 1, add 方法用来加上某个数值, sum 方法用来获取总和值。

代码清单 5-41 LongAdder 方式

```
final LongAdder adder = new LongAdder();
for (...)
    pool.submit(()->{
        while(...) {
            ...
            if(...) adder.increment();
        }
    });
long total = adder.sum();
```

注意: increment 方法不会返回原始值。使用它只会抹杀将总和值拆分为多个被加数所带来的性能提升。

AtomicLong 的实现方式是内部有个 value 变量, 当多线程并发自增, 自减时, 均通过 cas 指令从机器指令级别操作保证并发的原子性。

LongAccumulator 将这个思想带到了任意的累加操作中。在构造函数中, 你需要提供操作类型及其中立元素。要与新值相加, 需要调用 accumulate 方法。然后再调用 get 方法来获取当前值。以下代码与 LongAdder 的效果是一样的。

代码清单 5-42 LongAccumulator 方式

```
LongAccumulator adder = new LongAccumulator(Long::sum, 0);
//在某些线程中
adder.accumulator(value);
```

在 LongAccumulator 的内部, 包含 a1, a2, ..., an 等多个变量。每个变量都被初始化为中立元素(在我们的例子中即为 0)。

当调用 `accumulate` 方法累加值 `v` 时, 这些变量其中之一会自动被更新为 `ai=aiopv`, 其中 `op` 是以中缀形式表示的累加操作。在我们的示例中, 调用 `accumulate` 会对某个变量 `i` 计算 `ai=ai+v`。

Java8 中还添加了一个 `StampedLock` 类, 它用来实现乐观读。首先调用 `tryOptimisticRead` 方法, 此时会获得一个“印戳”。然后你读取值并检查“印戳”是否仍然有效 (例如, 没有其他线程已经获得了一个写锁)。如果有效, 你就可以使用这个值; 如果无效, 你会获得一个读锁 (它将会阻塞所有的写锁)。具体示例如下所示。

代码清单 5-43 `StampedLock` 使用示例

```
import java.util.concurrent.locks.StampedLock;

public class StampedLockDemo {
    private int size;
    private Object[] elements;
    private StampedLock lock = new StampedLock();

    public Object get(int n){
        long stamp = lock.tryOptimisticRead();
        Object[] currentElements = elements;
        int currentSize = size;
        if(!lock.validate(stamp)){//某个线程持有了一个写锁
            stamp = lock.readLock();//获得一个悲观锁
            currentElements = elements;
            currentSize = size;
            lock.unlockRead(stamp);
        }
        return n < currentSize ? currentElements[n]:null;
    }

    public static void main(String[] args){
        StampedLockDemo stampedLock = new StampedLockDemo();
        stampedLock.get(3);
    }
}
```

处理器通过以下几个步骤来使用原子操作。

- (1) 使用总线锁保证原子性: 如果多个处理器同时对共享变量进行读改写 操作, 那么共享变量就会被多个处理器同时进行操作, 这样读改写操作就不是原子的, 操作完之后共享变量的值会和期望的不一致。举个例子, 如果 `i=1`, 我们进行两次 `i++` 操作, 期望的结果是 3, 但是有可能结果是 2。原因可能是多个处理器同时从各自的缓存中读取变量 `i`, 分别进行加 1 操作, 然后分别写入系统内存中。那么, 想要保证读改写共享变量的操作是原子的, 就必须保证 CPU1 读改写共享变量的时候, CPU2 不能操作缓存了该共享变量内存地址的缓存。处理器使用总线锁来解决这个问题。所谓总线锁就是使用处理器提供的一个 `LOCK#` 信号, 当一个处理器在总线上输出此信号时, 其他处理器的请求将被阻塞住, 那么该处理器可以独占共享内存。

- (2) 使用缓存锁保证原子性：在同一时刻，我们只需保证对某个内存地址的操作是原子性即可，但总线锁定把 CPU 和内存之间的通信锁住了，这使得锁定期间，其他处理器不能操作其他内存地址的数据，所以总线锁定的开销比较大，目前处理器在某些场合下使用缓存锁定代替总线锁定来进行优化。

频繁使用的内存会缓存在处理器的 L1、L2 和 L3 高速缓存里，那么原子操作就可以直接在处理器内部缓存中进行，而不需要申明总线锁。所谓“缓存锁定”是指内存区域如果被缓存在处理器的缓存行中，并且在 LOCK 操作期间被锁定，那么当它执行锁操作回写到内存时，处理器不在总线上声言 LOCK#信号，而是修改内部的内存地址，并允许它的缓存一致性机制来保证操作的原子性，因为缓存一致性机制会阻止同时修改由两个以上处理器缓存的内存区域数据，当其他处理器回写已被锁定的缓存行的数据时，会使缓存行无效。

有两种情况处理器不会使用缓存锁定：

- (1) 当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行（cache line）时，则处理器会调用总线锁定。
- (2) 有些处理器不支持缓存锁定。

CAS 实现原子操作的三大问题

在 Java 并发包中有一些并发框架也使用了自旋 CAS 的方式来实现原子操作。比如 `LinkedTransferQueue` 类的 `Xfer` 方法。CAS 虽然很高效地解决了原子操作，但是 CAS 仍然存在三大问题。

- (1) ABA 问题。因为 CAS 需要在操作值的时候，检查值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是 A，变成了 B，又变成了 A，那么使用 CAS 进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA 问题的解决思路就是使用版本号。在变量前面追加了版本号，每次变量更新的时候把版本号加 1，那么 $A \rightarrow B \rightarrow A$ 就会变成 $1A \rightarrow 2B \rightarrow 3A$ 。从 Java5 开始，`Atomic` 包里提供了一个 `AtomicStampedReference` 来解决 ABA 问题。这个类的 `compareAndSet` 方法的作用是首先检查当前引用是否等于预期引用，并且检查当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。
- (2) 循环时间长、开销大：自旋 CAS 如果长时间不成功，会给 CPU 带来非常大的执行开销。如果 JVM 能支持处理器提供的 `pause` 指令，那么效率会有一定的提升。`pause` 指令有两个作用：第一，它可以延迟流水线执行指令，使 CPU 不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零；第二，它可以避免在推出循环的时候因内存顺序冲突（Memory order Violation）而引起 CPU 流水线被清空（CPU Pipeline Flush），从而提高 CPU 的执行效率。
- (3) 只能保证一个共享变量的原子操作。当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以使用锁。

5.4.2 并行容器

5.4.2.1 ConcurrentHashMap

在多线程环境中使用 Map，一般也可以使用 Collections 的 `synchronizedMap()` 方法得到一个线程安全的 Map。但是在高并发的情况，这个 Map 的性能表现不是最优的。由于 Map 是使用相当频繁的一个数据结构，因此 JDK 中便提供了一个专用于高并发的 Map 实现 `ConcurrentHashMap`。

`ConcurrentHashMap` 是 Java5 中支持高并发、高吞吐量的线程安全 `HashMap` 实现。`HashMap` 是非线程安全的，`ConcurrentHashMap` 具体是怎么实现线程安全的呢，肯定不可能是每个方法加 `synchronized`，那样就变成了 `HashTable`。`ConcurrentHashMap` 允许多个修改操作并发进行，其关键在于使用了锁分离技术。锁分离技术使用了多个锁来控制对 Hash 表的不同部分进行的修改。`ConcurrentHashMap` 内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个小的 Hash Table，它们都有自己的锁。只要多个修改操作发生在不同的段上，它们就可以并发进行。通过把整个 Map 分为 N 个 Segment（类似 `HashTable`），可以提供相同的线程安全，但是效率提升 N 倍，默认提升 16 倍。

有些方法需要跨段，比如 `size()` 和 `containsValue()`，它们可能需要锁定整个表而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。这里“按顺序”是很重要的，否则极有可能出现死锁，在 `ConcurrentHashMap` 内部，段数组是 `final` 的，并且其成员变量实际上也是 `final` 的，但是，仅仅是将数组声明为 `final` 的并不能保证数组成员也是 `final` 的，这需要通过实现上的保证，下面是段的不变性声明。这样做的好处是可以确保使用过程中不会出现死锁，因为获得锁的顺序是固定的。不变性使多线程编程占有很重要的地位。

代码清单 5-44 不变性定义

```
/**
 * The segments, each of which is a specialized hash table
 */
final Segment<K,V>[] segments;
```

`ConcurrentHashMap` 完全允许多个读操作并发进行，读操作并不需要加锁。如果使用传统的技术，如 `HashMap` 中的实现，如果允许可以在 Hash 链的中间添加或删除元素，读操作不加锁将得到不一致的数据。`ConcurrentHashMap` 实现技术是保证 `HashEntry` 几乎是不可变的。`HashEntry` 代表每个 Hash 链中的一个节点，其结构如下所示。

代码清单 5-45 HashEntry 实现

```
static final class HashEntry<K,V> {
    final K key;
    final int hash;
    volatile V value;
    final HashEntry<K,V> next;
}
```

可以看到除了 `value` 不是 `final` 的，其他值都是 `final` 的，这意味着不能从 hash 链的中间或尾部添加或删除节点，因为这需要修改 `next` 引用值，所有的节点的修改只能从头部开始。对于 put 操

作，可以一律添加到 Hash 链的头部。但是对于 remove 操作，可能需要从中间删除一个节点，这就需要将要删除节点的前面所有节点整个复制一遍，最后一个节点指向要删除结点的下一个结点。这在讲解删除操作时还会详述。为了确保读操作能够看到最新的值，将 value 设置成 volatile，这避免了加锁。

为了加快定位段以及段中 hash 槽的速度，每个段 hash 槽的个数都是 2^n ，这使得通过位运算就可以定位段和段中 hash 槽的位置。当并发级别为默认值 16 时，也就是段的个数，hash 值的高 4 位决定分配在哪个段中。但是我们也不要忘记《算法导论》给我们的教训：hash 槽的个数不应该是 2^n ，这可能导致 hash 槽分配不均，这需要对 hash 值重新再 hash 一次。下面是 Hash 算法的实现代码。

代码清单 5-46 Hash 实现代码

```
private static int hash(int h) {
    // Spread bits to regularize both segment and index locations,
    // using variant of single-word Wang/Jenkins hash.
    h += (h << 15) ^ 0xffffcd7d;
    h ^= (h >>> 10);
    h += (h << 3);
    h ^= (h >>> 6);
    h += (h << 2) + (h << 14);
    return h ^ (h >>> 16);
}
```

Hash 表的一个很重要方面就是如何解决 hash 冲突，ConcurrentHashMap 和 HashMap 使用相同的方式，都是将 hash 值相同的节点放在一个 hash 链中。与 HashMap 不同的是，ConcurrentHashMap 使用多个子 Hash 表，也就是段(Segment)。下面是 ConcurrentHashMap 的数据成员。

代码清单 5-47 ConcurrentHashMap 数据成员

```
public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>
    implements ConcurrentMap<K, V>, Serializable {
    /**
     * Mask value for indexing into segments. The upper bits of a
     * key's hash code are used to choose the segment.
     */
    final int segmentMask;
    /**
     * Shift value for indexing within segments.
     */
    final int segmentShift;
    /**
     * The segments, each of which is a specialized hash table
     */
    final Segment<K,V>[] segments;
}
```

所有的成员都是 Final 的，其中 segmentMask 和 segmentShift 主要是为了定位段，参见上面的

segmentFor 方法。每个 Segment 相当于一个子 Hash 表，它的数据成员如下所示。

代码清单 5-48 数据成员

```
static final class Segment<K,V> extends ReentrantLock implements Serializable{
    private static final long serialVersionUID = 2249069246763182397L;

    /**
     * The number of elements in this segment's region.
     */
    transient volatile int count;

    /**
     * Number of updates that alter the size of the table. This is
     * used during bulk-read methods to make sure they see a
     * consistent snapshot: If modCounts change during a traversal
     * of segments computing size or checking containsValue, the
     * we might have an inconsistent view of state so (usually)
     * must retry.
     */
    transient int modCount;

    /**
     * The table is rehashed when its size exceeds this threshold.
     * (The value of this field is always <tt>(int) (capacity *
     * loadFactor)</tt>.)
     */
    transient int threshold;

    /**
     * The per-segment table.
     */
    transient volatile HashEntry<K,V>[] table;

    /**
     * The load factor for the hash table. Even though this value
     * is same for all segments, it is replicated to avoid needing
     * links to outer object.
     * @serial
     */
    final float loadFactor;
}
```

count 用来统计该段数据的个数，它是 volatile，它用来协调修改和读取操作，以保证读取操作能够读取到几乎最新的修改。协调方式是这样的，每次修改操作做了结构上的改变，如增加/删除节点（修改节点的值不算结构上的改变），都要写 count 值，每次读取操作开始都要读取 count 的值。这利用了 Java5 中对 volatile 语义的增强，对同一个 volatile 变量的写和读存在 happens-before 关系。modCount 统计段结构改变的次数，主要是为了检测对多个段进行遍历过程中某个段是否发生改变，在讲述跨段操作时会还会详述。threshold 用来表示需要进行 rehash 的界限值。table 数组存储段中节点，每个数组元素是个 hash 链，用 HashEntry 表示。table 也是 volatile，这使得能够读取到最新的 table 值而不需要同步。loadFactor 表示负载因子。

ConcurrentHashMap 的整个删除操作是在持有段锁的情况下执行的，空白行之前的行主要是定

位到要删除的节点 e。接下来，如果不存在这个节点就直接返回 null，否则就要将 e 前面的结点复制一遍，尾结点指向 e 的下一个结点。e 后面的结点不需要复制，它们可以重用。

删除元素之前如图 5-6 所示。

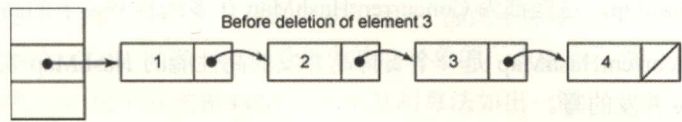


图5-6 删除元素前

删除元素 3 之后如图 5-7 所示。

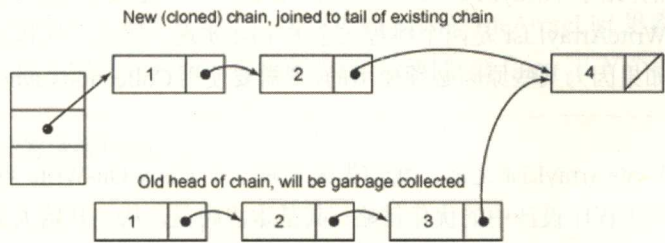


图5-7 删除元素后

整个 remove 实现并不复杂，但是需要注意如下几点。第一，当要删除的结点存在时，删除的最后一步操作要将 count 的值减一。这必须是最后一步操作，否则读取操作可能看不到之前对段所做的结构性修改。第二，remove 执行的开始就将 table 赋给一个局部变量 tab，这是因为 table 是 volatile 变量，读写 volatile 变量的开销很大。编译器也不能对 volatile 变量的读写做任何优化，直接多次访问非 volatile 实例变量没有多大影响，编译器会做相应优化。

ConcurrentHashMap 的 get 操作不需要锁。第一步是访问 count 变量，这是一个 volatile 变量，由于所有的修改操作在进行结构修改时都会在最后一步写 count 变量，通过这种机制保证 get 操作能够得到几乎最新的结构更新。对于非结构更新，也就是结点值的改变，由于 HashEntry 的 value 变量是 volatile 的，也能保证读取到最新的值。接下来就是对 hash 链进行遍历找到要获取的结点，如果没有找到，直接访问 null。对 hash 链进行遍历不需要加锁的原因在于链指针 next 是 final 的。但是头指针却不是 final 的，这是通过 getFirst(hash)方法返回，也就是存在 table 数组中的值。这使得 getFirst(hash)可能返回过时的头结点，例如，当执行 get 方法时，刚执行完 getFirst(hash)之后，另一个线程执行了删除操作并更新头结点，这就导致 get 方法中返回的头结点不是最新的。这是可以允许，通过对 count 变量的协调机制，get 能读取到几乎最新的数据，虽然可能不是最新的。要得到最新的数据，只有采用完全的同步。

最后，如果找到了所求的结点，判断它的值如果非空就直接返回，否则在有锁的状态下再读一次。理论上结点的值不可能为空，因为 put 的时候就进行了判断，如果为空就要抛 NullPointerException。空值的唯一源头就是 HashEntry 中的默认值，因为 HashEntry 中的 value 不是 final 的，非同步读取有可能读取到空值。仔细看下 put 操作的语句：tab[index] = new HashEntry<K,V>(key, hash, first, value)，在这条语句中，HashEntry 构造函数中对 value 的赋值以及对 tab[index]的赋值可能被重新排序，这就可能导致结点的值为空。这种情况应当很罕见，一旦发生这种情况，ConcurrentHashMap 采取的方式是在持有锁的情况下再读一遍，这能够保证读到最新的值，并且一定不会为空值。

我们分别使用 `ConcurrentHashMap` 和同步 `HashMap` 进行测试，就 `Map` 的高并发读写而言，`ConcurrentHashMap` 比 `HashMap` 快 1 倍。`ConcurrentHashMap` 之所以有如此高的吞吐量，得益于其内部实现进行了锁分离，同时，`CocurrentHashMap` 的 `get()` 操作也是无锁的，它的 `put()` 操作的锁粒度又小于同步的 `HashMap`，这些都为 `ConcurrentHashMap` 在多线程并发下的高性能提供了保证。

总的来说，`ConcurrentHashMap` 是一个支持高并发的高性能的 `HashMap` 实现，它支持完全并发的读以及一定程度并发的写。

5.4.2.2 CopyOnWriteArrayList

第 3 章我们重点介绍了 `ArrayList`，大家知道，`ArrayList` 不是线程安全的，在多线程环境下，`Vector` 或者 `CopyOnWriteArrayList` 是两个线程安全的 `List` 实现。因此，应该尽量避免在多线程环境下使用 `ArrayList`。如果因为某些原因必须使用的，则需要使用 `Collections.synchronizedList(List list)` 进行包装。

在了解 `CopyOnWriteArrayList` 之前，我们先来了解一下 `Copy-On-Write` 容器。`Copy-On-Write` 简称 `COW`，是一种用于程序设计中的优化策略。其基本思路是，从一开始大家都在共享同一个内容，当某个人想要修改这个内容的时候，才会真正把内容 `Copy` 出去形成一个新的内容然后再改，这是一种延时懒惰策略。从 `JDK1.5` 开始 `Java` 并发包里提供了两个使用 `CopyOnWrite` 机制实现的并发容器，它们是 `CopyOnWriteArrayList` 和 `CopyOnWriteArraySet`。

`CopyOnWrite` 容器非常有用，可以在非常多的并发场景中使用到。该容器是一种写时复制的容器，通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行 `Copy`，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对 `CopyOnWrite` 容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以 `CopyOnWrite` 容器也是一种读写分离的思想，读和写不同的容器。

总的来说，`CopyOnWrite` 容器有很多优点，但是同时也存在两个问题，即内存占用问题和数据一致性问题。所以在开发的需要注意一下。

(1) 内存占用问题。因为 `CopyOnWrite` 的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，旧的对象和新写入的对象（注意：在复制的时候只是复制容器里的引用，只是在写的时候会创建新对象添加到新容器里，而旧容器的对象还在使用，所以有两份对象内存）。如果这些对象占用的内存比较大，比如说 200M 左右，那么再写入 100M 数据进去，内存就会占用 300M，那么这个时候很有可能造成频繁的 `Yong GC` 和 `Full GC`。之前我们系统中使用了一个服务由于每晚使用 `CopyOnWrite` 机制更新大对象，造成了每晚 15 秒的 `Full GC`，应用响应时间也随之变长。

针对内存占用问题，可以通过压缩容器中的元素的方法来减少大对象的内存消耗，比如，如果元素全是 10 进制的数字，可以考虑把它压缩成 36 进制或 64 进制。或者不使用 `CopyOnWrite` 容器，而使用其他的并发容器，如 `ConcurrentHashMap`。

(2) 数据一致性问题。`CopyOnWrite` 容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的数据，马上能读到，请不要使用 `CopyOnWrite` 容器。

`CopOnWriteArrayList` 的内部实现与 `Vector` 不同。`CopOnWriteArrayList` 实现了当对象进行写操作的时候复制该对象的功能。如果进行的是读操作，则直接返回结果，注意整个操作过程中不进行同步操作。`CopyOnWriteArrayList` 很好地利用了对象的不变性，在没有对对象进行写操作前，由于对象未发生改变，因此不需要加锁。而在试图改变对象时，就像前面说的 COW 容器一样，总是先获取对象的一个副本，然后对副本进行修改，最后将副本写回，这个方式和 JVM 的标记复制算法如出一辙。

这种实现方式的核心思想是减少锁竞争，从而提高在高并发时的读取性能，但是这种做法却在一定程度上牺牲了写的性能。

`CopyOnWriteArrayList` 中 `add` 方法的实现（向 `CopyOnWriteArrayList` 里添加元素）如下所示，可以发现在添加的时候是需要加锁的，否则多线程写的时候会 `Copy` 出 N 个副本出来。

代码清单 5-49 `add` 方法源代码

```
/**
 * Appends the specified element to the end of this list.
 *
 * @param e element to be appended to this list
 * @return <tt>true</tt> (as specified by {@link Collection#add})
 */
public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        int len = elements.length;
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        newElements[len] = e;
        setArray(newElements);
        return true;
    } finally {
        lock.unlock();
    }
}
```

在每一次的 `add()` 方法中，`CopyOnWriteArrayList` 都是进行一次自我复制，同时，`add()` 操作也申请了锁，并不像 `get()` 方法那样。相对地，`Vector` 的 `add()` 方法则要快捷一些。因此，在高并发且以读为主的应用场景中，`CopyOnWriteArrayList` 要优于 `Vector`。但是，当写操作也很频繁时，`CopyOnWriteArrayList` 效率并不高，所以应该优先使用 `Vector`。

再来看一下读的操作（`get` 方法）。读的时候不需要加锁，如果读的时候有多个线程正在向 `CopyOnWriteArrayList` 添加数据，读还是会读到旧的数据，因为写的时候不会锁住旧的 `CopyOnWriteArrayList`。

代码清单 5-50 `get` 方法源代码

```
public E get(int index) {
```

```

        return get(getArray(), index);
    }

```

可以看到，作为一个线程安全的实现，`CopOnWriteArrayList` 的 `get()` 方法并没有任何锁操作。而对比 `Vector` 的 `get()` 实现，`Vector` 使用了同步关键字，所有的 `get()` 操作都必须先取得对象锁才能进行。在高并发的情况下，大量的锁竞争会拖累系统性能。

只要了解了 `CopyOnWrite` 机制，我们就可以模仿着实现各种 `CopyOnWrite` 容器，并且在不同的应用场景中使用。JDK 中并没有提供 `CopyOnWriteMap`，我们可以参考 `CopyOnWriteArrayList` 来实现一个，基本代码如下所示。

代码清单 5-51 CopyOnWriteMap 实现

```

import java.util.Collection;
import java.util.Map;
import java.util.Set;

public class CopyOnWriteMap<K, V> implements Map<K, V>, Cloneable {
    private volatile Map<K, V> internalMap;

    public CopyOnWriteMap() {
        internalMap = new HashMap<K, V>();
    }

    public V put(K key, V value) {

        synchronized (this) {
            Map<K, V> newMap = new HashMap<K, V>(internalMap);
            V val = newMap.put(key, value);
            internalMap = newMap;
            return val;
        }
    }

    public V get(Object key) {
        return internalMap.get(key);
    }

    public void putAll(Map<? extends K, ? extends V> newData) {
        synchronized (this) {
            Map<K, V> newMap = new HashMap<K, V>(internalMap);
            newMap.putAll(newData);
            internalMap = newMap;
        }
    }
}

```

从前面的描述可以知道，`CopyOnWrite` 并发容器用于读多写少的并发场景，比如白名单，黑名单，商品类目的访问和更新场景。假设我们有一个搜索网站，用户在这个网站的搜索框中输入

关键字搜索内容，但是我们需要能够屏蔽某些关键字。这些不能被搜索的关键字会被放在一个黑名单当中，黑名单每天晚上更新一次。当用户搜索时，会检查当前关键字是否在黑名单当中，如果在，则提示不能搜索。实现代码如下所示，我们使用了前面定义的 CopyOnWriteMap 容器。

代码清单 5-52 黑名单服务实现

```
import java.util.Map;
import com.ifeve.book.forkjoin.CopyOnWriteMap;
/**
 * 黑名单服务
 *
 * @author fangtengfei
 *
 */
public class BlackListServiceImpl {

    private static CopyOnWriteMap<String, Boolean> blackListMap = new CopyOnWriteMap<String, Boolean>(
        1000);

    public static boolean isBlackList(String id) {
        return blackListMap.get(id) == null ? false : true;
    }

    public static void addBlackList(String id) {
        blackListMap.put(id, Boolean.TRUE);
    }

    /**
     * 批量添加黑名单
     *
     * @param ids
     */
    public static void addBlackList(Map<String, Boolean> ids) {
        blackListMap.putAll(ids);
    }
}
```

代码很简单，但是使用 CopyOnWriteMap 需要注意两件事情：

- (1) 减少扩容开销。根据实际需要，初始化 CopyOnWriteMap 的大小，避免写时 CopyOnWriteMap 扩容的开销。
- (2) 使用批量添加。因为每次添加，容器每次都会进行复制，所以减少添加次数，可以减少容器的复制次数。如使用上面代码里的 addBlackList 方法。

和 List 相似，并发 Set 组件也有一个 CopOnWriteArrayList，它实现了 Set 接口，并且是线程安全的。它的内部实现完全依赖于 CopOnWriteArrayList，因此，它的特性和 CopOnWriteArrayList 完全一致，适用于读写多少的高并发场合。在需要并发写的场合，则可以使用 Collections 的方法

`public static <T> Set<T> synchronizedSet(Set<T> s)`得到一个线程安全的 Set。

5.4.3 非阻塞队列

5.4.3.1 非阻塞算法

基于锁的算法会带来一些活跃度失败的风险。如果线程在持有锁的时候因为阻塞 I/O，页面错误，或其他原因发生延迟，很可能所有的线程都不能前进了。一个线程的失败或挂起不应该影响其他线程的失败或挂起，这样的算法成为非阻塞（nonblocking）算法；如果算法的每一个步骤中都有有一些线程能够继续执行，那么这样的算法称为锁自由（lock-free）算法。在线程间使用 CAS 进行协调，这样的算法如果能构建正确的话，它既是非阻塞的，又是锁自由的。非竞争的 CAS 总是能够成功，如果多个线程以一个 CAS 竞争，总会有一个胜出并前进。非阻塞算法堆死锁和优先级倒置有“免疫性”（它们可能会出现饥饿和活锁，因为它们允许重进入）。

非阻塞算法属于并发算法，它们可以安全地派生它们的线程，不通过锁定派生，而是通过低级的原子性的硬件原生形式，例如比较和交换。非阻塞算法的设计与实现极为困难，但是它们能够提供更好的吞吐率，对生存问题（例如死锁和优先级反转）也能提供更好的防御。原子变量类向用户提供了这些底层级原语，也能够当做“更佳的 volatile 变量”使用，同时提供了整数类和对对象引用的原子化更新操作。

下面的示例代码实现了非阻塞堆栈，其中的 `push()` 和 `pop()` 操作在结构上希望在提交工作的时候，底层假设没有失效。`push()` 方法观察当前最顶的节点，构建一个新节点放在堆栈上，然后，如果最顶端的节点在初始观察之后没有变化，那么就安装新节点。如果 CAS 失败，意味着另一个线程已经修改了堆栈，那么过程就会重新开始。

代码清单 5-53 使用 Treiber 算法的非阻塞堆栈

```
AtomicReference<Node<E>> head = new AtomicReference<Node<E>>();
```

```
public void push(E item) {
    Node<E> newHead = new Node<E>(item);
    Node<E> oldHead;
    do {
        oldHead = head.get();
        newHead.next = oldHead;
    } while (!head.compareAndSet(oldHead, newHead));
}
```

```
public E pop() {
    Node<E> oldHead;
    Node<E> newHead;
    do {
        oldHead = head.get();
        if (oldHead == null)
            return null;
        newHead = oldHead.next;
    } while (!head.compareAndSet(oldHead, newHead));
}
```



```

        return oldHead.item;
    }

    static class Node<E> {
        final E item;
        Node<E> next;

        public Node(E item) { this.item = item; }
    }
}

```

在轻度到中度的争用情况下，非阻塞算法的性能会超越阻塞算法，因为 CAS 的多数时间都在第一次尝试时就成功，而发生争用时的开销也不涉及线程挂起和上下文切换，只多了几个循环迭代。没有争用的 CAS 要比没有争用的锁便宜得多（这句话肯定是真的，因为没有争用的锁涉及 CAS 加上额外的处理），而争用的 CAS 比争用的锁获取涉及更短的延迟。

在高度争用的情况下（即有多个线程不断争用一个内存位置的时候），基于锁的算法开始提供比非阻塞算法更好的吞吐率，因为当线程阻塞时，它就会停止争用，耐心地等候轮到自己，从而避免了进一步争用。但是，这么高的争用程度并不常见，因为多数时候，线程会把线程本地的计算与争用共享数据的操作分开，从而给其他线程使用共享数据的机会。（这么高的争用程度也表明需要重新检查算法，朝着更少共享数据的方向努力。）

目前为止的示例（计数器和堆栈）都是非常简单的非阻塞算法，一旦掌握了在循环中使用 CAS，就可以容易地模仿它们。对于更复杂的数据结构，非阻塞算法要比这些简单示例复杂得多，因为修改链表、树或哈希表可能涉及对多个指针的更新。CAS 支持对单一指针的原子性条件更新，但是不支持两个以上的指针。所以，要构建一个非阻塞的链表、树或哈希表，需要找到一种方式，可以用 CAS 更新多个指针，同时不会让数据结构处于不一致的状态。

在链表的尾部插入元素，通常涉及对两个指针的更新：“尾”指针总是指向列表中的最后一个元素，“下一个”指针从过去的最后一个元素指向新插入的元素。因为需要更新两个指针，所以需要两个 CAS。在独立的 CAS 中更新两个指针带来了两个需要考虑的潜在问题：如果第一个 CAS 成功，而第二个 CAS 失败，会发生什么？如果其他线程在第一个和第二个 CAS 之间企图访问链表，会发生什么？

对于非复杂数据结构，构建非阻塞算法的“技巧”是确保数据结构总处于一致的状态（甚至包括在线程开始修改数据结构和它完成修改之间），还要确保其他线程不仅能够判断出第一个线程已经完成了更新还是处在更新的中途，还能够判断出如果第一个线程走向 AWOL，完成更新还需要什么操作。如果线程发现了处在更新中途的数据结构，它就可以“帮助”正在执行更新的线程完成更新，然后再进行自己的操作。当第一个线程回来试图完成自己的更新时，会发现不再需要了，返回即可，因为 CAS 会检测到帮助线程的干预（在这种情况下，是建设性的干预）。

这种“帮助邻居”的要求，对于让数据结构免受单个线程失败的影响，是必需的。如果线程发现数据结构正处在被其他线程更新的中途，就会等候其他线程完成更新，那么如果其他线程在操作中途失败，这个线程就可能永远等候下去。即使不出现故障，这种方式也会提供糟糕的性能，因为新到达的线程必须放弃处理器，导致上下文切换，或者等到自己的时间片过期。

非阻塞算法要比基于锁的算法复杂得多。开发非阻塞算法是相当专业的训练，而且要证明算法的正确也极为困难。但是在 Java 版本之间并发性能上的众多改进来自对非阻塞算法的采用，而且随着并发性能变得越来越重要，可以预见在 Java 平台的未来发行版中，会使用更多的非阻塞算法。

5.4.3.2 ConcurrentLinkedQueue

在 Java 多线程应用中，队列的使用率很高，多数生产消费模型的首选数据结构就是队列(先进先出)。Java 提供的线程安全的队列方式可以分为阻塞队列和非阻塞队列，其中阻塞队列的典型例子是 BlockingQueue，非阻塞队列的典型例子是 ConcurrentLinkedQueue，在实际应用中要根据实际需要选用阻塞队列或者非阻塞队列。不论哪种实现，都继承自 Queue 接口。

ConcurrentLinkedQueue 是 Queue 的一个安全实现。Queue 中元素按 FIFO 原则进行排序，采用 CAS 操作来保证元素的一致性，当我们添加一个元素的时候，它会添加到队列的尾部，当我们获取一个元素时，它会返回队列头部的元素。它采用了“wait-free⁵”算法来实现。由于 LinkedBlockingQueue 实现是线程安全的，实现了先进先出等特性，是作为生产者消费者的首选，LinkedBlockingQueue 可以指定容量，也可以不指定，不指定的话，默认最大是 Integer.MAX_VALUE，其中主要用到 put 和 take 方法，put 方法在队列满的时候会阻塞直到有队列成员被消费，take 方法在队列空的时候会阻塞，直到有队列成员被放进来。

ConcurrentLinkedQueue 由 head 节点和 tail 节点组成，每个节点 (Node) 由节点元素 (item) 和指向下一个节点的引用(next)组成，节点与节点之间就是通过这个 next 关联起来，从而组成一张链表结构的队列。默认情况下 head 节点存储的元素为空，tail 节点等于 head 节点，如代码 private transient volatile Node<E> tail = head;

入队列就是将入队节点添加到队列的尾部。为了方便理解入队时队列的变化，以及 head 节点和 tail 节点的变化，每添加一个节点我就做了一个队列的快照图，如图 5-8 所示。

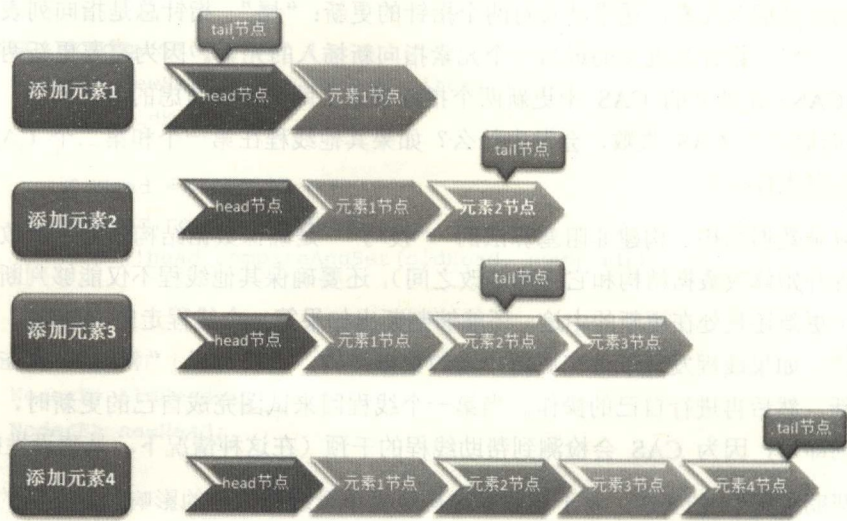


图5-8 出入队列流程图

⁵ 该算法在 Michael & Scott 算法上进行了一些修改。

- (1) 添加元素 1。队列更新 head 节点的 next 节点为元素 1 节点。又因为 tail 节点默认情况下等于 head 节点，所以它们的 next 节点都指向元素 1 节点；
- (2) 添加元素 2。队列首先设置元素 1 节点的 next 节点为元素 2 节点，然后更新 tail 节点指向元素 2 节点；
- (3) 添加元素 3，设置 tail 节点的 next 节点为元素 3 节；
- (4) 添加元素 4，设置元素 3 的 next 节点为元素 4 节点，然后将 tail 节点指向元素 4 节点。

入队主要做两件事情，第一是将入队节点设置成当前队列尾节点的下一个节点。第二是更新 tail 节点，如果 tail 节点的 next 节点不为空，则将入队节点设置成 tail 节点，如果 tail 节点的 next 节点为空，则将入队节点设置成 tail 的 next 节点，所以 tail 节点不总是尾节点。

上面的分析让我们从单线程入队的角度来理解入队过程，但是多个线程同时进行入队情况就变得更加复杂，因为可能会出现其他线程插队的情况。如果有一个线程正在入队，那么它必须先获取尾节点，然后设置尾节点的下一个节点为入队节点，但这时可能有另外一个线程插队了，那么队列的尾节点就会发生变化，这时当前线程要暂停入队操作，然后重新获取尾节点。让我们再通过源码来详细分析下它是如何使用 CAS 算法来入队的。

代码清单 5-54 ConcurrentLinkedQueue 源代码

```
public boolean offer(E e){
    if (e == null) throw new NullPointerException();
    //入队前，创建一个入队节点
    Node<E> n = new Node<E>(e);
    retry:
    //死循环，入队不成功反复入队。
    for (;;) {
        //创建一个指向 tail 节点的引用
        Node<E> t = tail;
        //p 用来表示队列的尾节点，默认情况下等于 tail 节点。
        Node<E> p = t;
        for (int hops = 0; ; hops++) {
            //获得 p 节点的下一个节点。
            Node<E> next = succ(p);
            //next 节点不为空，说明 p 不是尾节点，需要更新 p 后在将它指向 next 节点
            if (next != null) {
                //循环了两次及其以上，并且当前节点还是不等于尾节点
                if (hops > HOPS && t != tail)
                    continue retry;
                p = next;
            }
            //如果 p 是尾节点，则设置 p 节点的 next 节点为入队节点。
            else if (p.casNext(null, n)) {
                //如果 tail 节点有大于等于 1 个 next 节点，则将入队节点设置成 tair 节点，更新失败了也没关系，因为失败了表示有其他线程成功更新了 tair 节点。
                if (hops >= HOPS)
                    casTail(t, n); // 更新 tail 节点，允许失败
            }
        }
    }
}
```

```
return true;
}
// p 有 next 节点,表示 p 的 next 节点是尾节点,则重新设置 p 节点
else {
    p = succ(p);
}
}
}
}
```

从源代码角度来看整个入队过程主要做两件事情。第一是定位出尾节点，第二是使用 CAS 算法将入队节点设置成尾节点的 next 节点，如不成功则重试。

第一步定位尾节点。tail 节点并不总是尾节点，所以每次入队都必须先通过 tail 节点来找到尾节点，尾节点可能就是 tail 节点，也可能是 tail 节点的 next 节点。代码中循环体中的第一个 if 就是判断 tail 是否有 next 节点，有则表示 next 节点可能是尾节点。获取 tail 节点的 next 节点需要注意的是 p 节点等于 p 的 next 节点的情况，只有一种可能就是 p 节点和 p 的 next 节点都等于空，表示这个队列刚初始化，正准备添加第一次节点，所以需要返回 head 节点。

第二步设置入队节点为尾节点。p.casNext(null, n)方法用于将入队节点设置为当前队列尾节点的 next 节点，p 如果是 null 表示 p 是当前队列的尾节点，如果不为 null 表示有其他线程更新了尾节点，则需要重新获取当前队列的尾节点。

出队列的就是从队列里返回一个节点元素，并清空该节点对元素的引用。让我们通过每个节点出队的快照来观察下 head 节点的变化，如图 5-9 所示。

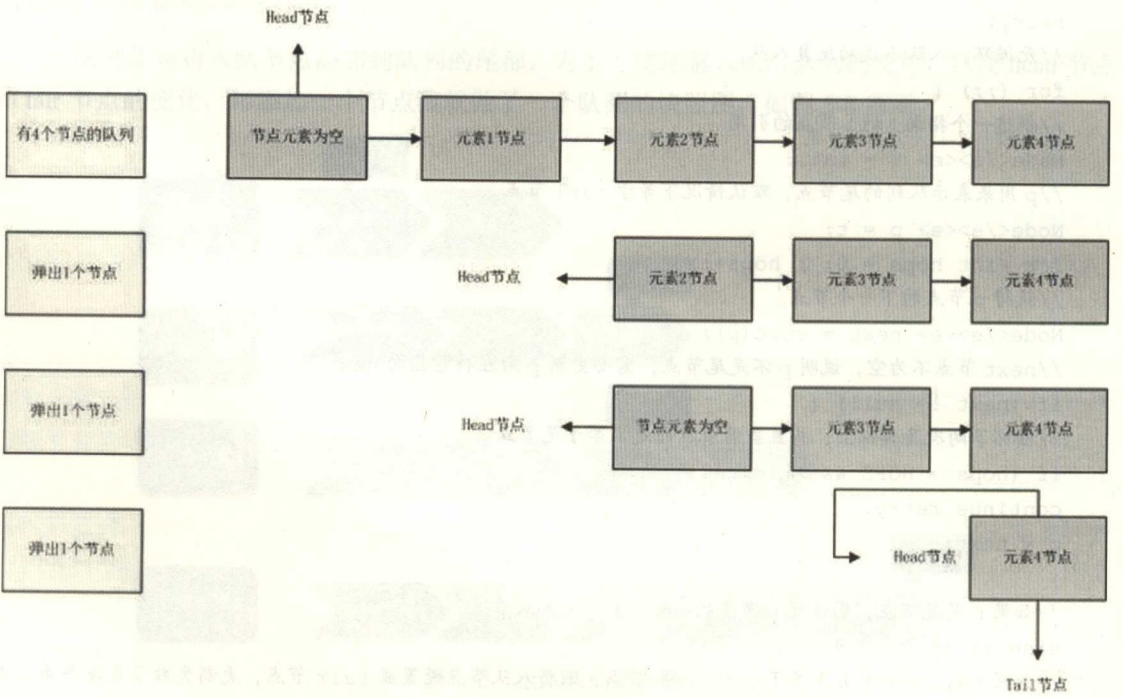


图5-9 队列变化图

从上图可知，并不是每次出队时都更新 head 节点，当 head 节点里有元素时，直接弹出 head

节点里的元素，而不会更新 head 节点。只有当 head 节点里没有元素时，出队操作才会更新 head 节点。这种做法也是通过 hops 变量来减少使用 CAS 更新 head 节点的消耗，从而提高出队效率。

从上面的介绍我们知道，ConcurrentLinkedQueue 是一个适用于高并发场景下的队列。它通过无锁的方式，实现了高并发状态下的高性能，通常，ConcurrentLinkedQueue 的性能要好于 BlockingQueue。使用以下代码测试 ConcurrentLinkedQueue 以及 LinkedBlockingQueue 的性能。

代码清单 5-55 ConcurrentLinkedQueue 测试

```
import java.util.concurrent.ConcurrentLinkedQueue;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ConcurrentLinkedQueueTest {
    private static ConcurrentLinkedQueue<Integer> queue = new ConcurrentLinkedQueue
<Integer>();
    private static int count = 2; // 线程个数
    //CountDownLatch, 一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。
    private static CountDownLatch latch = new CountDownLatch(count);

    public static void main(String[] args) throws InterruptedException {
        long timeStart = System.currentTimeMillis();
        ExecutorService es = Executors.newFixedThreadPool(4);
        ConcurrentLinkedQueueTest.offer();
        for (int i = 0; i < count; i++) {
            es.submit(new Poll());
        }
        latch.await(); //使得主线程(main)阻塞直到 latch.countDown()为零才继续执行
        System.out.println("cost time " + (System.currentTimeMillis() - timeStart)
+ "ms");
        es.shutdown();
    }

    /**
     * 生产
     */
    public static void offer() {
        for (int i = 0; i < 100000; i++) {
            queue.offer(i);
        }
    }

    /**
     * 消费
     */
}
```

```
* @author 林计钦
* @version 1.0 2013-7-25 下午 05:32:56
*/
static class Poll implements Runnable {
    public void run() {
        // while (queue.size()>0) {
        while (!queue.isEmpty()) {
            System.out.println(queue.poll());
        }
        latch.countDown();
    }
}
```

ConcurrentLinkedQueue 和其他并发集合一样，它把一个元素放入到队列的线程的优先级高于对元素的访问和移除的线程。

总的来说，使用非阻塞队列的好处是允许多线程操作共同的队列时不需要额外的同步消耗，另外就是队列会自动平衡负载，即那边（生产与消费两边）处理快了就会被阻塞掉，从而减少两边的处理速度差距。

5.4.4 阻塞队列

与 ConcurrentLinkedQueue 的使用场景不同的是，在 Java 的 Concurrent 包中，添加了阻塞队列 BlockingQueue。BlockingQueue 的主要功能并不是在于提升高并发时的队列性能，而在于简化多线程间的数据共享。BlockingQueue 是一个队列，一个队列在数据结构中所起的作用大致如图 5-10 所示。

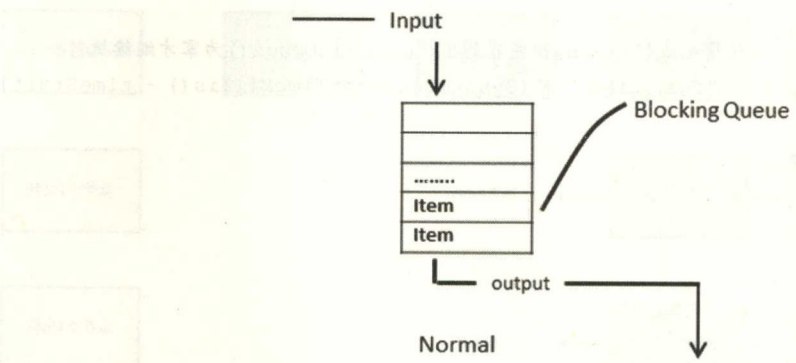


图5-10 队列的作用

从上图我们可以很清楚地看到，通过一个共享的队列，可以使得数据由队列的一端输入，从另外一端输出。

常用的队列主要有以下两种，当然通过不同的实现方式，还可以延伸出很多不同类型的队列，DelayQueue 就是其中的一种。

- 先进先出（FIFO）：先插入的队列的元素也最先出队列，类似于排队功能。从某种程度

上来说这种队列也体现了一种公平性。

■ **后进先出 (LIFO)**：后插入队列的元素最先出队列，这种队列优先处理最近发生的事件。

多线程环境中，通过队列可以很容易实现数据共享，比如经典的“生产者”和“消费者”模型中，通过队列可以很便利地实现两者之间的数据共享。假设我们有若干生产者线程，另外又有若干个消费者线程。如果生产者线程需要把准备好的数据共享给消费者线程，利用队列的方式来传递数据，就可以很方便地解决他们之间的数据共享问题。但如果生产者和消费者在某个时间段内，万一发生数据处理速度不匹配的情况呢？理想情况下，如果生产者产出数据的速度大于消费者消费的速度，并且当生产出来的数据累积到一定程度的时候，那么生产者必须暂停等待一下（阻塞生产者线程），以便等待消费者线程把累积的数据处理完毕，反之亦然。然而，在 concurrent 包发布以前，在多线程环境下，我们每个程序员都必须去自己控制这些细节，尤其还要兼顾效率和线程安全，而这会给我们的程序带来不小的复杂度。好在此时，强大的 concurrent 包横空出世了，而他也给我们带来了强大的 BlockingQueue。（在多线程领域：所谓阻塞，在某些情况下会挂起线程（即阻塞），一旦条件满足，被挂起的线程又会自动被唤醒）

图 5-11 和图 5-12 演示了 BlockingQueue 的两个常见阻塞场景。

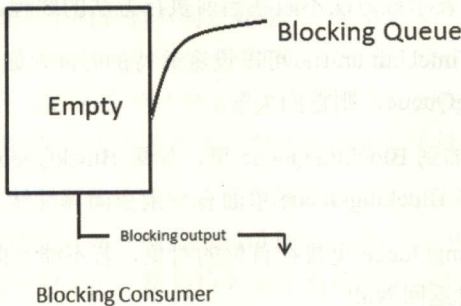


图5-11 常见阻塞场景

如上图所示：当队列中没有数据的情况下，消费者端的所有线程都会被自动阻塞（挂起），直到有数据放入队列。

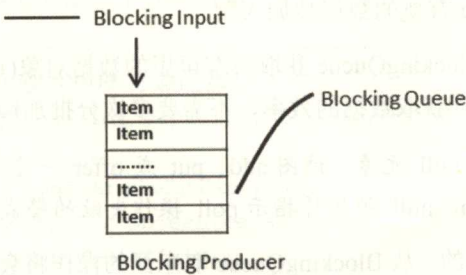


图5-12 自动阻塞场景

如上图所示：当队列中填满数据的情况下，生产者端的所有线程都会被自动阻塞（挂起），直到队列中有空的位置，线程被自动唤醒。

这也是我们在多线程环境下，为什么需要 BlockingQueue 的原因。作为 BlockingQueue 的使用

者，我们再也不需要关心什么时候需要阻塞线程，什么时候需要唤醒线程，因为这一切 `BlockingQueue` 都给你一手包办了。

`BlockingQueue` 的典型应用场景是在生产者-消费者模式中，生产者总是将产品放入 `BlockingQueue` 队列，而消费者从队列中取出产品消费，从而实现数据共享。

`BlockingQueue` 提供一种读写阻塞等待的机制，即如果消费者速度较快，则 `BlockingQueue` 可能被清空，此时，消费线程再试图从 `BlockingQueue` 读取数据时就会被阻塞。反之，如果生产线程较快，则 `BlockingQueue` 可能会被装满，此时，生产线程再试图向 `BlockingQueue` 队列中装入数据时，便会被阻塞等待。

`BlockingQueue` 类族最常用的应用场景是多线程间的数据共享，如果需要高性能的队列，可以使用 `ConcurrentLinkedQueue`。`BlockingQueue` 不光实现了一个完整队列所具有的基本功能，同时在多线程环境下，他还自动管理了多线程间的自动等待与唤醒功能，从而使得程序员可以忽略这些细节，关注更高级的功能。

`BlockingQueue` 的核心方法如下。

- `offer(object)`: 将 `object` 加到 `BlockingQueue` 里，如果 `BlockingQueue` 有足够空间，则返回 `true`，否则返回 `false`，表示该方法不阻塞当前执行方法的线程。
- `offer(E o, long timeout, TimeUnit unit)`: 可以设定等待的时间，如果在指定的时间内，还不能往队列中加入 `BlockingQueue`，则返回失败。
- `put(object)`: 把 `object` 加到 `BlockingQueue` 里，如果 `BlockingQueue` 没有足够空间，则调用此方法的线程被阻断直到 `BlockingQueue` 里面有空闲空间再继续。
- `poll(time)`: 取走 `BlockingQueue` 里排在首位的对象，若不能立即取出，则可以等 `time` 参数规定的时间，取不到时返回 `Null`。
- `poll(long timeout, TimeUnit unit)`: 从 `BlockingQueue` 取出一个队首的对象，如果在指定时间内，队列一旦有数据可取，则立即返回队列中的数据。超时后依然没有取得数据则返回失败。
- `take()`: 取走 `BlockingQueue` 里排在首位的对象，若 `BlockingQueue` 为空，阻断进入等待队列直到 `BlockingQueue` 有新的数据被加入。
- `drainTo()`: 一次性从 `BlockingQueue` 获取所有可用的数据对象(还可以指定获取数据的个数)，通过此方法，可以提升获取数据的效率，不需要多次分批加锁或释放锁。

注意：`BlockingQueue` 不接受 `null` 元素。试图 `add`、`put` 或 `offer` 一个 `null` 元素时，某些实现会抛出 `NullPointerException`。`null` 被用作指示 `poll` 操作失败的警戒值。

如果 `BlockingQueue` 是空的，从 `BlockingQueue` 取东西的操作将会被阻断进入等待状态，直到 `BlockingQueue` 进了东西才会被唤醒，同样，如果 `BlockingQueue` 是满的，任何试图往里存东西的操作也会被阻断进入等待状态，直到 `BlockingQueue` 里有空间时才会被唤醒继续操作。

`BlockingQueue` 接口提供了五种主要的实现包括 `ArrayBlockingQueue`、`LinkedBlockingQueue`、`PriorityBlockingQueue`、`DelayQueue` 和 `SynchronousQueue`，

5.4.5.1 ArrayBlockingQueue

它是一种基于数组的阻塞队列实现，在 `ArrayBlockingQueue` 内部，维护了一个定长数组，用于缓存队列中的数据对象。此外，`ArrayBlockingQueue` 内部还保持着两个整型变量，分别标识着队列的头部和尾部在数组中的位置。在创建 `ArrayBlockingQueue` 时，还可以控制对象的内部锁是否采用公平锁，默认采用非公平锁。

`ArrayBlockingQueue` 是一种规定大小的 `BlockingQueue`，其构造函数必须带一个 `int` 参数来指明其大小。其所含的对象是以 FIFO（先入先出）顺序排序的。基于数组的阻塞队列实现，在 `ArrayBlockingQueue` 内部，维护了一个定长数组，以便缓存队列中的数据对象，这是一个常用的阻塞队列，除了一个定长数组外，`ArrayBlockingQueue` 内部还保存着两个整型变量，分别标识着队列的头部和尾部在数组中的位置。`ArrayBlockingQueue` 在生产者放入数据和消费者获取数据，都是共用同一个锁对象，由此也意味着两者无法真正并行运行，这点尤其不同于 `LinkedBlockingQueue`；按照实现原理来分析，`ArrayBlockingQueue` 完全可以采用分离锁，从而实现生产者和消费者操作的完全并行运行。Doug Lea 之所以没这样去做，也许是因为 `ArrayBlockingQueue` 的数据写入和获取操作已经足够轻巧，以至于引入独立的锁机制，除了给代码带来额外的复杂性外，其在性能上完全占不到任何便宜。`ArrayBlockingQueue` 和 `LinkedBlockingQueue` 间还有一个明显的不同之处在于，前者在插入或删除元素时不会产生或销毁任何额外的对象实例，而后者则会生成一个额外的 `Node` 对象。这在长时间内需要高效并发地处理大批量数据的系统中，其对于 GC 的影响还是存在一定的区别。而在创建 `ArrayBlockingQueue` 时，我们还可以控制对象的内部锁是否采用公平锁，默认采用非公平锁。

`ArrayBlockingQueue` 是一个由数组支持的有界阻塞队列。此队列按 FIFO（先进先出）原则对元素进行排序。队列的头部是在队列中存在时间最长的元素，队列的尾部是在队列中存在时间最短的元素。新元素插入到队列的尾部，队列检索操作则是从队列头部开始获得元素。

这是一个典型的“有界缓存区”，固定大小的数组在其中保持生产者插入的元素和使用者提取的元素。一旦创建了这样的缓存区，就不能再增加其容量。试图向已满队列中放入元素会导致放入操作受阻塞；试图从空队列中检索元素将导致类似阻塞。

`ArrayBlockingQueue` 创建的时候需要指定容量 `capacity`（可以存储的最大的元素个数，因为它不会自动扩容）。其中一个构造方法如下代码所示。

代码清单 5-56 ArrayBlockingQueue 构造函数

```
public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.items = (E[]) new Object[capacity];
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull = lock.newCondition();
}
```

`ArrayBlockingQueue` 类中定义的变量有：

```
/** The queued items */
private final E[] items;
```

```

/** items index for next take, poll or remove */
private int takeIndex;
/** items index for next put, offer, or add. */
private int putIndex;
/** Number of items in the queue */
private int count;

/*
 * Concurrency control uses the classic two-condition algorithm
 * found in any textbook.
 */
/** Main lock guarding all access */
private final ReentrantLock lock;
/** Condition for waiting takes */
private final Condition notEmpty;
/** Condition for waiting puts */
private final Condition notFull;

```

使用数组 `items` 来存储元素，由于是循环队列，使用 `takeIndex` 和 `putIndex` 来标记 `put` 和 `take` 的位置。可以看到，该类中只定义了一个锁 `ReentrantLock`，定义两个 `Condition` 对象：`notEmpty` 和 `notFull`，分别用来对 `take` 和 `put` 操作进行所控制。注：本文主要讲解 `put()` 和 `take()` 操作，其他方法类似。

`put(E e)` 方法的源码如下。进行 `put` 操作之前，必须获得锁并进行加锁操作，以保证线程安全性。加锁后，若发现队列已满，则调用 `notFull.await()` 方法，如当前线程陷入等待。直到其他线程 `take` 走某个元素后，会调用 `notFull.signal()` 方法来激活该线程。激活之后，继续下面的插入操作。

代码清单 5-57 put 方法源代码

```

/**
 * Inserts the specified element at the tail of this queue, waiting
 * for space to become available if the queue is full.
 *
 */
public void put(E e) throws InterruptedException {
    //不能存放 null 元素
    if (e == null) throw new NullPointerException();
    final E[] items = this.items; //数组队列
    final ReentrantLock lock = this.lock;
    //加锁
    lock.lockInterruptibly();
    try{
        try{
            //当队列满时，调用 notFull.await() 方法，使该线程阻塞。
            //直到 take 掉某个元素后，调用 notFull.signal() 方法激活该线程。
            while (count == items.length)
                notFull.await();
        }catch(InterruptedException ie){

```



```

        notFull.signal(); // propagate to non-interrupted thread
        throw ie;
    }
    //把元素 e 插入到队尾
    insert(e);
}finally{
    //解锁
    lock.unlock();
}
}
}

```

insert(E e)方法如下所示。

代码清单 5-58 inset 方法源代码

```

/**
 * Inserts element at current put position, advances, and signals.
 * Call only when holding lock.
 */
private void insert(E x){
    items[putIndex] = x;
    //下标加 1 或者等于 0
    putIndex = inc(putIndex);
    ++count; //计数加 1
    //若有 take() 线程陷入阻塞, 则该操作激活 take() 线程, 继续进行取元素操作。
    //若没有 take() 线程陷入阻塞, 则该操作无意义。
    notEmpty.signal();
}

/**
 *Circularly increment i.
 */
final int inc(int i){
    //此处可以看到使用了循环队列
    return (++i == items.length)? 0 : i;
}

```

take()方法代码如下。take 操作和 put 操作相反, 故不作详细介绍。

代码清单 5-59 take 方法源代码

```

public E take() throws InterruptedException{
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly(); //加锁
    try{
        try{
            //当队列空时, 调用 notEmpty.await() 方法, 使该线程阻塞。
            //直到 take 掉某个元素后, 调用 notEmpty.signal() 方法激活该线程。
            while (count == 0)
                notEmpty.await();
        }catch (InterruptedException ie){

```

```

        notEmpty.signal();//propagate to non-interrupted thread
        throw ie;
    }
    //取出队头元素
    E x = extract();
    return x;
}finally{
    lock.unlock();//解锁
}
}

```

`extract()` 方法如下所示。

代码清单 5-60 `extract` 方法源代码

```

/**
 * Extracts element at current take position, advances, and signals.
 *Call only when holding lock.
 */
private E extract(){
    final E[] items = this.items;
    E x = items[takeIndex];
    items[takeIndex] = null;
    takeIndex = inc(takeIndex);
    --count;
    notFull.signal();
    return x;
}

```

从上面的源代码可以看出,在进行 `put` 和 `take` 操作时共用同一个锁对象。也即是说, `put` 和 `take` 无法并行执行。

5.4.5.2 `LinkedBlockingQueue`

这是一个基于链表的阻塞队列,与 `ArrayListBlockingQueue` 类似,它实现了 `BlockingQueue` 接口,内部也维持着一个数据缓冲队列(该队列由一个链表构成),当生产者往队列中放入一个数据时,队列会从生产者手中获取数据,并缓存在队列内部,而生产者立即返回;只有当队列缓冲区达到最大值缓存容器时(`LinkedBlockingQueue` 可以通过构造函数指定该值,默认不限制大小),才会阻塞生产者队列,知道消费者从队列中消费掉一个数据,生产者线程才会被唤醒。

由于 `LinkedBlockingQueue` 实现是线程安全的,实现了先进先出等特性,是作为生产者消费者的首选, `LinkedBlockingQueue` 可以指定容量,也可以不指定,不指定的话,默认最大是 `Integer.MAX_VALUE`,其中主要用到 `put` 和 `take` 方法, `put` 方法在队列满的时候会阻塞直到有队列成员被消费, `take` 方法在队列空的时候会阻塞,直到有队列成员被放进来。

代码清单 5-61 `BlockingQueue` 测试

```

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;

```



```

import java.util.concurrent.Executors;
import java.util.concurrent.LinkedBlockingQueue;

/**
 * 多线程模拟实现生产者/消费者模型
 */
public class BlockingQueueTest2 {
    /**
     *
     * 定义装苹果的篮子
     *
     */
    public class Basket {
        // 篮子，能够容纳 3 个苹果
        BlockingQueue<String> basket = new LinkedBlockingQueue<String>(3);

        // 生产苹果，放入篮子
        public void produce() throws InterruptedException {
            // put 方法放入一个苹果，若 basket 满了，等到 basket 有位置
            basket.put("An apple");
        }

        // 消费苹果，从篮子中取走
        public String consume() throws InterruptedException {
            // take 方法取出一个苹果，若 basket 为空，等到 basket 有苹果为止(获取并移除此队列的头部)
            return basket.take();
        }
    }

    // 定义苹果生产者
    class Producer implements Runnable {
        private String instance;
        private Basket basket;

        public Producer(String instance, Basket basket) {
            this.instance = instance;
            this.basket = basket;
        }

        public void run() {
            try {
                while (true) {
                    // 生产苹果
                    System.out.println("生产者准备生产苹果: " + instance);
                    basket.produce();
                    System.out.println("!生产者生产苹果完毕: " + instance);
                    // 休眠 300ms
                    Thread.sleep(300);
                }
            }
        }
    }
}

```

```
    }  
    } catch (InterruptedException ex) {  
        System.out.println("Producer Interrupted");  
    }  
}  
}  
  
// 定义苹果消费者  
class Consumer implements Runnable {  
    private String instance;  
    private Basket basket;  
  
    public Consumer(String instance, Basket basket) {  
        this.instance = instance;  
        this.basket = basket;  
    }  
  
    public void run() {  
        try {  
            while (true) {  
                // 消费苹果  
                System.out.println("消费者准备消费苹果: " + instance);  
                System.out.println(basket.consume());  
                System.out.println("!消费者消费苹果完毕: " + instance);  
                // 休眠 1000ms  
                Thread.sleep(1000);  
            }  
        } catch (InterruptedException ex) {  
            System.out.println("Consumer Interrupted");  
        }  
    }  
}  
  
public static void main(String[] args) {  
    BlockingQueueTest2 test = new BlockingQueueTest2();  
  
    // 建立一个装苹果的篮子  
    Basket basket = test.new Basket();  
  
    ExecutorService service = Executors.newCachedThreadPool();  
    Producer producer = test.new Producer("生产者 001", basket);  
    Producer producer2 = test.new Producer("生产者 002", basket);  
    Consumer consumer = test.new Consumer("消费者 001", basket);  
    service.submit(producer);  
    service.submit(producer2);  
    service.submit(consumer);  
}
```


`LinkedBlockingQueue` 是大小不定的 `BlockingQueue`，若其构造函数带一个规定大小的参数，生成的 `BlockingQueue` 有大小限制，若不带大小参数，所生成的 `BlockingQueue` 的大小由 `Integer.MAX_VALUE` 来决定。其所含的对象是以 FIFO 顺序排序的。基于链表的阻塞队列，同 `ArrayListBlockingQueue` 类似，其内部也维持着一个数据缓冲队列（该队列由一个链表构成），当生产者将数据放入队列中时，队列会从生产者手中获取数据，并缓存在队列内部，而生产者立即返回；只有当队列缓冲区达到最大值缓存容量时（`LinkedBlockingQueue` 可以通过构造函数指定该值），才会阻塞生产者队列，直到消费者从队列中消费掉一份数据，生产者线程会被唤醒，反之对于消费者这端的处理也基于同样的原理。而 `LinkedBlockingQueue` 之所以能够高效的处理并发数据，还因为其对于生产者端和消费者端分别采用了独立的锁来控制数据同步，这也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。作为开发者，我们需要注意的是，如果构造一个 `LinkedBlockingQueue` 对象，而没有指定其容量大小，`LinkedBlockingQueue` 会默认一个类似无限大小的容量（`Integer.MAX_VALUE`），这样的话，如果生产者的速度一旦大于消费者的速度，也许还没有等到队列满阻塞产生，系统内存就有可能已被消耗殆尽了。`ArrayBlockingQueue` 和 `LinkedBlockingQueue` 是两个最普通也是最常用的阻塞队列，一般情况下，在处理多线程间的生产者消费者问题，使用这两个类足矣。

`LinkedBlockingQueue` 和 `ArrayBlockingQueue` 比较起来，它们背后所用的数据结构不一样，导致 `LinkedBlockingQueue` 的数据吞吐量要大于 `ArrayBlockingQueue`，但在线程数量很大时其性能的可预见性低于 `ArrayBlockingQueue`。

基于链表的阻塞队列，同 `ArrayListBlockingQueue` 类似，其内部也维持着一个数据缓冲队列（该队列由一个链表构成），当生产者往队列中放入一个数据时，队列会从生产者手中获取数据，并缓存在队列内部，而生产者立即返回；只有当队列缓冲区达到最大值缓存容量时（`LinkedBlockingQueue` 可以通过构造函数指定该值），才会阻塞生产者队列，直到消费者从队列中消费掉一份数据，生产者线程会被唤醒，反之对于消费者这端的处理也基于同样的原理。而 `LinkedBlockingQueue` 之所以能够高效的处理并发数据，还因为其对于生产者端和消费者端分别采用了独立的锁来控制数据同步，这也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。

作为开发者，我们需要注意的是，如果构造一个 `LinkedBlockingQueue` 对象，而没有指定其容量大小，`LinkedBlockingQueue` 会默认一个类似无限大小的容量（`Integer.MAX_VALUE`），这样的话，如果生产者的速度一旦大于消费者的速度，也许还没有等到队列满阻塞产生，系统内存就有可能已被消耗殆尽了。

`LinkedBlockingQueue` 类中定义的变量如下所示。

代码清单 5-62 `LinkedBlockingQueue` 中定义的变量

```
/** The capacity bound, or Integer.MAX_VALUE if none */
private final int capacity;
/** Current number of elements */
private final AtomicInteger count = new AtomicInteger(0);
/** Head of linked list */
private transient Node<E> head;
/** Tail of linked list */
```



```

private transient Node<E> last;
/** Lock held by take, poll, etc */
private final ReentrantLock takeLock = new ReentrantLock();
/** Wait queue for waiting takes */
private final Condition notEmpty = takeLock.newCondition();
/** Lock held by put, offer, etc */
private final ReentrantLock putLock = new ReentrantLock();
/** Wait queue for waiting puts */
private final Condition notFull = putLock.newCondition();

```

该类中定义了两个 `ReentrantLock` 锁, 即 `putLock` 和 `takeLock`, 它们分别用于 `put` 端和 `take` 端。也就是说, 生成端和消费端各自独立拥有一把锁, 避免了读 (`take`) 写 (`put`) 时互相竞争锁的情况。

代码清单 5-63 put 方法源代码

```

/**
 * Inserts the specified element at the tail of this queue, waiting if
 * necessary for space to become available.
 */
public void put(E e) throws InterruptedException{
    if (e == null) throw new NullPointerException();
    //Note: convention in all put/take/etc is to preset local var
    //holding count negative to indicate failure unless set.
    int c = -1;
    final ReentrantLock putLock = this.putLock;
    final AtomicInteger count = this.count;
    putLock.lockInterruptibly(); //加 putLock 锁
    try{
        /**
         * Note that count is used in wait guard even though it is
         * not protected by lock. This works because count can
         * only decrease at this point (all other puts are shut
         * out by lock), and we (or some other waiting put) are
         * signalled if it ever changes from
         * capacity. Similarly for all other uses of count in
         * other wait guards.
         */
        //当队列满时, 调用 notFull.await() 方法释放锁, 陷入等待状态。
        //有两种情况会激活该线程
        //第一、某个 put 线程添加元素后, 发现队列有空余, 就调用 notFull.signal() 方法激活阻塞线程
        //第二、take 线程取元素时, 发现队列已满。则其取出元素后, 也会调用 notFull.signal() 方法激活阻塞线程
        while (count.get() == capacity) {
            notFull.await();
        }
        //把元素 e 添加到队列中 (队尾)
        enqueue(e);
        c = count.getAndIncrement();
    }
}

```



```
//发现队列未满,调用 notFull.signal() 激活阻塞的 put 线程(可能存在)
if (c + 1 < capacity)
notFull.signal();
}finally{
putLock.unlock();
}
if (c == 0){
//队列空,说明已经有 take 线程陷入阻塞,故调用 signalNotEmpty 激活阻塞的 take 线程
}signalNotEmpty();
}
}
```

enqueue(E e)方法如下所示。

代码清单 5-64 enqueue 方法源代码

```
/**
 * Creates a node and links it at end of queue.
 * @param x the item
 */
05.private void enqueue(E x) {
// assert putLock.isHeldByCurrentThread();
last = last.next = new Node<E>(x);
08.}
```

take()方法代码如下。take 操作和 put 操作相反,故不作详细介绍。

代码清单 5-65 take 方法源代码

```
public E take() throws InterruptedException{
    E x;
    int c = -1;
    final AtomicInteger count = this.count;
    final ReentrantLock takeLock = this.takeLock;
    takeLock.lockInterruptibly();
    try{
        while(count.get() == 0){
            notEmpty.await();
        }
        x = dequeue();
        c = count.getAndDecrement();
        if (c > 1)
            notEmpty.signal();
    } finally {
        takeLock.unlock();
    }
    if (c == capacity)
        signalNotFull();
    return x;
}
```

dequeue()方法如下所示。

代码清单 5-66 dequeue 方法源代码

```
/**
 * Removes a node from head of queue.
 * @return the node
 */
private E dequeue() {
    //assert takeLock.isHeldByCurrentThread();
    Node<E> h = head;
    Node<E> first = h.next;
    h.next = h; // help GC
    head = first;
    E x = first.item;
    first.item = null;
    return x;
}
```

小结：take 和 put 操作各有一把锁，可并行读取。

5.4.5.3 DelayQueue

假设我们有这么个工作场景。服务器里面有很多客户端的连接，空闲一段时间之后需要逐一关闭。一种可行的方法是使用一个后台线程，遍历所有连接，然后挨个检查、关闭。这种笨笨的办法简单好用，但是对象数量过多时，可能存在性能问题，检查间隔时间不好设置，间隔时间过大，影响精确度，多小则存在效率问题，而且做不到按超时的时间顺序处理。总的来说，这种场景使用 DelayQueue 最适合了。

DelayQueue 是一个 BlockingQueue，其特化的参数是 Delayed。Delayed 扩展了 Comparable 接口，比较的基准为延时的时间值，Delayed 接口的实现类 getDelay 的返回值应为固定值（final）。DelayQueue 内部是使用 PriorityQueue 实现的。所以我们可以用一个公式概括这些组成因素，即 DelayQueue = BlockingQueue + PriorityQueue + Delayed。可以这么说，DelayQueue 是一个使用优先队列（PriorityQueue）实现的 BlockingQueue，优先队列的比较基准值是时间。

DelayQueue 内部的实现使用了一个优先队列。当调用 DelayQueue 的 offer 方法时，把 Delayed 对象加入到优先队列 q 中，代码如下所示。

代码清单 5-67 DelayQueue 的 Offer 方法源代码

```
public boolean offer(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        E first = q.peek();
        q.offer(e);
        if (first == null || e.compareTo(first) < 0)
            available.signalAll();
        return true;
    }
```



```

    } finally {
        lock.unlock();
    }
}

```

DelayQueue 的 take 方法，把优先队列 q 的 first 拿出来 (peek)，如果没有达到延时阈值，则进行 await 处理，代码如下所示。

代码清单 5-68 DelayQueue 的 take 方法源代码

```

public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        for (;;) {
            E first = q.peek();
            if (first == null) {
                available.await();
            } else {
                long delay = first.getDelay(TimeUnit.NANOSECONDS);
                if (delay > 0) {
                    long tl = available.awaitNanos(delay);
                } else {
                    E x = q.poll();
                    assert x != null;
                    if (q.size() != 0)
                        available.signalAll(); // wake up other takers
                    return x;
                }
            }
        }
    } finally {
        lock.unlock();
    }
}

```

DelayQueue 中的元素只有当其指定的延迟时间到了，才能够从队列中获取到该元素。DelayQueue 是一个没有大小限制的队列，因此往队列中插入数据的操作（生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞。DelayQueue 使用场景较少，但都相当巧妙。

我们来模拟一个示例，假设模拟一个考试，考试时间为 120 分钟，30 分钟后才可交卷，当时间到了，或学生都交完卷了考试结束。用 DelayQueue 存储考生 (Student 类)，每一个考生都有自己的名字和完成试卷的时间，Teacher 线程对 DelayQueue 进行监控，收取完成试卷小于 120 分钟的学生的试卷。当考试时间 120 分钟到时，先关闭 Teacher 线程，然后强制 DelayQueue 中还存在的考生交卷。每一个考生交卷都会进行一次 countDownLatch.countDown()，当 countDownLatch.await() 不再阻塞说明所有考生都交卷了，判断结束后结束考试。代码如下所示。

代码清单 5-69 DelayQueue 示例

```

import java.util.Iterator;
import java.util.Random;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.Delayed;
import java.util.concurrent.DelayQueue;
import java.util.concurrent.Delayed;
import java.util.concurrent.TimeUnit;

public class examinationClass {
    public static void main(String[] args) throws InterruptedException {
        // TODO Auto-generated method stub
        int studentNumber = 20;
        CountDownLatch countDownLatch = new CountDownLatch(studentNumber+1);
        DelayQueue< Student> students = new DelayQueue<Student>();
        Random random = new Random();
        for (int i = 0; i < studentNumber; i++) {
            students.put(new Student("student"+(i+1), 30+random.nextInt(120),countDownLatch));
        }
        Thread teacherThread =new Thread(new Teacher(students));
        students.put(new EndExam(students, 120,countDownLatch,teacherThread));
        teacherThread.start();
        countDownLatch.await();
        System.out.println(" 考试时间到，全部交卷！ ");
    }
}

class Student implements Runnable,Delayed{

    private String name;
    private long workTime;
    private long submitTime;
    private boolean isForce = false;
    private CountDownLatch countDownLatch;

    public Student(){}

    public Student(String name,long workTime,CountDownLatch countDownLatch){
        this.name = name;
        this.workTime = workTime;
        this.submitTime = TimeUnit.NANOSECONDS.convert(workTime, TimeUnit.NANOSECONDS)+
System.nanoTime();
        this.countDownLatch = countDownLatch;
    }

    @Override
    public int compareTo(Delayed o) {
        // TODO Auto-generated method stub

```



```

        if(o == null || ! (o instanceof Student)) return 1;
        if(o == this) return 0;
        Student s = (Student)o;
        if (this.workTime > s.workTime) {
            return 1;
        }else if (this.workTime == s.workTime) {
            return 0;
        }else {
            return -1;
        }
    }

    @Override
    public long getDelay(TimeUnit unit) {
        // TODO Auto-generated method stub
        return unit.convert(submitTime - System.nanoTime(), TimeUnit.NANOSECONDS);
    }

    @Override
    public void run() {
        // TODO Auto-generated method stub
        if (isForce) {
            System.out.println(name + " 交卷, 希望用时" + workTime + "分钟"+" ,实际用
时 120 分钟" );
        }else {
            System.out.println(name + " 交卷, 希望用时" + workTime + "分钟"+" ,实际用
时 "+workTime +" 分钟");
        }
        countDownLatch.countDown();
    }

    public boolean isForce() {
        return isForce;
    }

    public void setForce(boolean isForce) {
        this.isForce = isForce;
    }
}

class EndExam extends Student{

    private DelayQueue<Student> students;
    private CountDownLatch countDownLatch;
    private Thread teacherThread;

    public EndExam(DelayQueue<Student> students, long workTime, CountDownLatch

```

```
countDownLatch, Thread teacherThread) {
    super("强制收卷", workTime, countDownLatch);
    this.students = students;
    this.countDownLatch = countDownLatch;
    this.teacherThread = teacherThread;
}

@Override
public void run() {
    // TODO Auto-generated method stub

    teacherThread.interrupt();
    Student tmpStudent;
    for (Iterator<Student> iterator2 = students.iterator(); iterator2.
hasNext();) {
        tmpStudent = iterator2.next();
        tmpStudent.setForce(true);
        tmpStudent.run();
    }
    countDownLatch.countDown();
}

}

class Teacher implements Runnable{

    private DelayQueue<Student> students;
    public Teacher(DelayQueue<Student> students){
        this.students = students;
    }

    @Override
    public void run() {
        // TODO Auto-generated method stub
        try {
            while(!Thread.interrupted()){
                students.take().run();
            }
        } catch (Exception e) {
            // TODO: handle exception
            e.printStackTrace();
        }
    }

}
```


在我们第4章提到过的Hbase中,Lease就是以—个DelayQueue存放的。比如说—个当Client扫描数据表的时候,首先会找到扫描startkey所在的Region,构建RegionScanner,将通过RegionScanner加入—个delayQueue,以及维护—个scannerId到RegionScanner的Map。HRegionServer启动的时候会起—个Leases的守护线程,时刻监听delayQueue的元素,当有过期时将其从Map中移除,从而达到Leases的作用。

5.4.5.4 PriorityBlockingQueue

PriorityBlockingQueue类似于LinkedBlockingQueue,但其所含对象的排序不是FIFO,而是依据对象的自然排序顺序或者是构造函数所带的Comparator决定的顺序。—个无界的阻塞队列,它使用与类PriorityQueue相同的顺序规则,并且提供了阻塞检索的操作。虽然此队列逻辑上是无界的,但是由于资源被耗尽,所以试图执行添加操作可能会失败(导致OutOfMemoryError)。此类不允许使用null元素。依赖自然顺序的优先级队列也不允许插入不可比较的对象(因为这样做会抛出ClassCastException)。

PriorityBlockingQueue是基于优先级的阻塞队列(优先级的判断通过构造函数传入的Comparator对象来决定),但需要注意的是PriorityBlockingQueue并不会阻塞数据生产者,而只会在没有可消费的数据时,阻塞数据的消费者。因此使用的时候要特别注意,生产者生产数据的速度绝对不能快于消费者消费数据的速度,否则时间—长,会最终耗尽所有的可用堆内存空间。在实现PriorityBlockingQueue时,内部控制线程同步的锁采用的是公平锁。

下面这个示例(JDK8不支持)我们对比了返回的Entity,可以确定Priority。

代码清单 5-70 PriorityBlockingQueue 示例程序

```
import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.PriorityBlockingQueue;
import java.util.concurrent.TimeUnit;

public class PriorityBlockingQueueDemo {
    static Random r=new Random(47);

    public static void main(String args[]){
        final PriorityBlockingQueue q= new PriorityBlockingQueue();
        ExecutorService se=Executors.newCachedThreadPool();
        //execute producer
        se.execute(new Runnable(){
            public void run() {
                int i=0;
                while(true){
                    q.put(new PriorityEntity(r.nextInt(10),i++));
                    try {
                        TimeUnit.MILLISECONDS.sleep(r.nextInt(1000));
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                    }
                }
            }
        });
    }
}
```

```

        e.printStackTrace();
    }
}

//execute consumer
se.execute(new Runnable(){

    public void run() {
        while(true){
            try {
                System.out.println("take-- "+q.take()+" left:-- ["+q.toString()+""]");
                try {
                    TimeUnit.MILLISECONDS.sleep(r.nextInt(1000));
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
});

try {
    TimeUnit.SECONDS.sleep(5);
} catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

System.out.println("shutdown");
}

}

class PriorityEntity implements Comparable<PriorityEntity> {
    private static int count=0;
    private int id=count++;
    private int priority;
    private int index=0;

    public PriorityEntity(int _priority,int _index) {
        this.priority = _priority;
        this.index=_index;
    }

    public String toString(){
        return id+"# [index="+index+" priority="+priority+"]";
    }
}

```



```

    }

    //数字小, 优先级高
    public int compareTo(PriorityEntity o) {
        return this.priority > o.priority ? 1
            : this.priority < o.priority ? -1 : 0;
    }

    //数字大, 优先级高
    public int compareTo(PriorityTask o) {
        return this.priority < o.priority ? 1
            : this.priority > o.priority ? -1 : 0;
    }
}

```

注意, `PriorityBlockingQueue` 里面存储的对象必须是实现 `Comparable` 接口。队列通过这个接口的 `Compare` 方法确定对象的 `Priority`, 具体规则是当与其他对象比较时, 如果 `Compare` 方法返回负数, 那么在队列里面的优先级就比较高。

5.4.5.5 SynchronousQueue

`SynchronousQueue` 是一种特殊的 `BlockingQueue`, 对其的操作必须是放和取交替完成的。它是一种无缓冲的等待队列, 类似于无中介的直接交易, 这一点有点像原始社会中的生产者和消费者, 生产者拿着产品去集市销售给产品的最终消费者, 而消费者必须亲自去集市找到所要商品的直接生产者, 如果一方没有找到合适的目标, 那么对不起, 大家都在集市等待。相对于有缓冲的 `BlockingQueue` 来说, 这样的设计方式减少了一个中间经销商的环节 (缓冲区), 如果有经销商, 生产者直接把产品批发给经销商, 而无须在意的经销商最终会将这些产品卖给那些消费者, 由于经销商可以库存一部分商品, 因此相对于直接交易模式, 总体来说采用中间经销商的模式会吞吐量高一些 (可以批量买卖); 但另一方面, 又因为经销商的引入, 使得产品从生产者到消费者中间增加了额外的交易环节, 单个产品的及时响应性能可能会降低。

从上面的描述我们可以得出, `SynchronousQueue` 是这样一种阻塞队列, 它的每个 `put` 必须等待一个 `take`, 反之亦然。同步队列没有任何内部容量, 甚至连一个队列的容量都没有。除非另一个线程试图移除某个元素, 否则也不能 (使用任何方法) 添加元素; 也不能迭代队列, 因为其中没有元素可用于迭代。队列的头是尝试添加到队列中的首个已排队线程元素, 如果没有已排队线程, 则不添加元素并且头为 `Null`。

`SynchronousQueue` 内部没有容量, 但是由于一个插入操作总是对应一个移除操作, 反过来同样需要满足。那么一个元素就不会在 `SynchronousQueue` 里面长时间停留, 一旦有了插入线程和移除线程, 元素很快就从插入线程移交给移除线程。也就是说这更像是一种信道 (管道), 资源从一个方向快速传递到另一方向。

需要特别说明的是, 尽管元素在 `SynchronousQueue` 内部不会 “停留”, 但是并不意味着 `SynchronousQueue` 内部没有队列。实际上 `SynchronousQueue` 维护者线程队列, 也就是插入线程或者移除线程在不同同时存在的时候就会有线程队列。既然有队列, 同样就有公平性和非公平性特性, 公平性保证正在等待的插入线程或者移除线程以 `FIFO` 的顺序传递资源。显然这是一种快速

传递元素的方式，也就是说在这种情况下元素总是以最快的方式从插入者（生产者）传递给移除者（消费者），这在多任务队列中是最快处理任务的方式。

声明一个 `SynchronousQueue` 有两种不同的方式，即公平模式和非公平模式，它们之间有着不太一样的行为。

- 如果采用公平模式：`SynchronousQueue` 会采用公平锁，并配合一个 FIFO 队列来阻塞多余的生产者和消费者，从而体系整体的公平策略；
- 但如果是非公平模式（`SynchronousQueue` 默认）：`SynchronousQueue` 采用非公平锁，同时配合一个 LIFO 队列来管理多余的生产者和消费者，而后一种模式，如果生产者和消费者的处理速度有差距，则很容易出现饥渴的情况，即可能有某些生产者或者是消费者的数据永远都得不到处理。

我们来看一个 `SynchronousQueue` 的简单使用示例，代码如下所示。

代码清单 5-71 `SynchronousQueue` 示例程序

```
import java.util.Random;
import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.TimeUnit;

public class SynchronousQueueDemo {
    public static void main(String[] args) {
        SynchronousQueue<Integer> queue = new SynchronousQueue<Integer>();
        new Customer(queue).start();
        new Product(queue).start();
    }

    static class Product extends Thread{
        SynchronousQueue<Integer> queue;
        public Product(SynchronousQueue<Integer> queue){
            this.queue = queue;
        }
        @Override
        public void run(){
            while(true){
                int rand = new Random().nextInt(1000);
                System.out.println("生产了一个产品: "+rand);
                System.out.println("等待三秒后运送出去...");
                try {
                    TimeUnit.SECONDS.sleep(3);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                queue.offer(rand);
            }
        }
    }
}
```



```

static class Customer extends Thread{
    SynchronousQueue<Integer> queue;
    public Customer(SynchronousQueue<Integer> queue){
        this.queue = queue;
    }
    @Override
    public void run(){
        while(true){
            try {
                System.out.println("消费了一个产品:"+queue.take());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("-----");
        }
    }
}

```

输出如下所示。

代码清单 5-72 SynchronousQueue 示例程序运行输出

生产了一个产品: 618

等待三秒后运送出去...

生产了一个产品: 765

等待三秒后运送出去...

消费了一个产品: 618

生产了一个产品: 433

等待三秒后运送出去...

消费了一个产品: 765

SynchronousQueue 有几点注意。

- (1) 它是一种阻塞队列，其中每个 put 必须等待一个 take，反之亦然。同步队列没有任何内部容量，甚至连一个队列的容量都没有。
- (2) 它是线程安全的，是阻塞的。
- (3) 不允许使用 null 元素。
- (4) 公平排序策略是指调用 put 的线程之间，或 take 的线程之间。公平排序策略可以查考 ArrayBlockingQueue 中的公平策略。

5.4.5.6 LinkedTransferQueue

前面一节里介绍的 BlockingQueue 是针对读取或者写入锁定整个队列，所以在比较繁忙的时候各种锁比较耗时。JDK7 的并发包里新增一个队列集合类 LinkedTransferQueue，该类继承自

TransferQueue 接口，TransferQueue 继承了 BlockingQueue（BlockingQueue 又继承了 Queue）并扩展了一些新方法。BlockingQueue（和 Queue）是 Java 5 中加入的接口，它是指这样的一个队列：当生产者向队列添加元素但队列已满时，生产者会被阻塞；当消费者从队列移除元素但队列为空时，消费者会被阻塞。TransferQueue 则更进一步，生产者会一直阻塞直到所添加到队列的元素被某一个消费者所消费（不仅仅是添加到队列里就完事）。新添加的 transfer 方法用来实现这种约束。顾名思义，阻塞就是发生在元素从一个线程 transfer 到另一个线程的过程中，它有效地实现了元素在线程之间的传递（以建立 Java 内存模型中的 happens-before 关系的方式）。

TransferQueue 还包括了其他的一些方法：两个 tryTransfer 方法，一个是非阻塞的，另一个带有 timeout 参数设置超时。还有两个辅助方法 hasWaitingConsumer() 和 getWaitingConsumerCount()。

LinkedTransferQueue 类在使用 volatile 变量时，用一种追加字节的方式来优化队列出队和入队的性能。LinkedTransferQueue 利用 CompareAndSwap 进行一个无阻塞的队列，针对每一个操作进行处理。从数据结构角度来看，LinkedTransferQueue 类的内部保持着一个栈，基本单位是 Node，根据 hasData 区分里面有两种元素，要么是 Data 要么是 Reservation，不会同时存在，并且有一个变量 Head 指向最前面的 Node，没东西则是 Null。

LinkedTransferQueue 类的完整存取过程分成两部分：

- 原节点与新节点的比较，代码如下所示。

代码清单 5-73 原节点与新节点的比较

```
for (;;) { // restart on append race
    for (Node h = head, p = h; p != null;) { // 如果头结点为空则跳过，非空进去找第一个可用节点
        boolean isData = p.isData;
        Object item = p.item;
        if (item != p && (item != null) == isData) { // 判断原节点可用性，如 data 的 item 应该是数值，如果是 null 则表明用过了
            if (isData == haveData) // 两个节点是相同类型，不用 match 了，去下一步
                break;
            if (p.casItem(item, e)) { // 节点不同类型，match 成功，更改原节点 item，表明不可用
                for (Node q = p; q != h;) { // 什么，我居然不是 head 节点了？我要让它指向我！
                    Node n = q.next; // update by 2 unless singleton
                    if (head == h && casHead(h, n == null ? q : n)) {
                        h.forgetNext();
                        break;
                    }
                } // advance and retry
                if ((h = head) == null || (q = h.next) == null || !q.isMatched())
                    break; // unless slack < 2
            }
        }
        LockSupport.unpark(p.waiter); // 根据原节点的类型，reservation 则叫人收货，data 则叫 null 收货
        return LinkedTransferQueue.<E>cast(item); // 根据原节点的类型，reservation 则返回 null，data 则返回数据
    }
}
```



```

Node n = p.next; // 下一个节点
p = (p != n) ? n : (h = head); // Use head if p offlist
}

```

上面的方法，重点目标是为了找出第一个可用节点，如果是 null 则跳过，如果与进来的节点相同（本来就有 data，还放 data）也跳过，如果不同（本来是 data，现在是 reservation，返回 data 值/本来是 reservation，现在是 data，叫人来收货，返回 reservation 值=空）。

■ 处理节点

如果比较失败了就会进入这个环节，然后把新节点放进栈内，并根据参数决定立刻返回或者等待返回。代码如下所示。

代码清单 5-74 处理节点

```

if (how != NOW){//No matches available
    if (s == null)
        s = new Node(e, haveData);
    Node pred = tryAppend(s, haveData); //尝试添加新 node
    if (pred == null)
        continue retry; // 不成功则重试整个过程
    if (how != ASYNC)
        return awaitMatch(s, pred, e, (how == TIMED), nanos); //根据参数，等不等
    别人放数据，拿数据，等多久
}
return e; // not waiting

```

LinkedTransferQueue 类内部采用的是一种非常不同的队列，即松弛型双重队列（Dual Queues with Slack）。松弛的意思是说，它的 head 和 tail 节点相较于其他并发队列要求上更放松，构造它的目的是减少 CAS 操作的次数（相应的会增加 next 域的引用次数），举个例子：某个瞬间 tail 指向的节点后面已经有 6 个节点了（以下图借用源码的注释），而其他并发队列真正的尾节点最多只能是 tail 的下一个节点。收缩的方式是大部分情况下不会用 tail 的 next 来设置 tail 节点，而是第一次收缩 N 个 next(N>=2)，然后查看能否 2 个一次来收缩 tail。双重是指有两种类型相互对立的节点 (Node.isData==false || true)，并且我理解的每种节点都有三种状态：

- (1) INIT（节点构造完成，刚进入队列的状态）；
- (2) MATCHED（节点备置为“满足”状态，即入队节点标识的线程成功取得或者传递了数据）；
- (3) CANCELED（节点被置为取消状态，即入队节点标识的线程因为超时或者中断决定放弃等待）。

在队列中已有元素的情况下，调用 transfer 方法，可以确保队列中被传递元素之前的所有元素都能被处理。LinkedTransferQueue 实际上是 ConcurrentLinkedQueue、SynchronousQueue（公平模式）和 LinkedBlockingQueue 的超集。而且 LinkedTransferQueue 更好用，因为它不仅仅综合了这几个类的功能，同时也提供了更高效的实现。

前面说过，LinkedTransferQueue 类使用一个内部类类型来定义队列的头节点和尾节点，而这

个内部类 `PaddedAtomicReference` 相对于父类 `AtomicReference` 只做了一件事情，就是将共享变量追加到 64 字节。一个对象的引用占用了 4 个字节，它追加了 15 个变量（共占 60 个字节），再加上父类的 `value` 变量，一共 64 个字节。这种设计的原理是基于 CPU 的原理，处理器的 L1、L2 或 L3 缓存的高速缓存行是 64 个字节宽，不支持部分填充缓存行，这意味着，如果队列的头节点和尾节点都不足 64 字节的话，处理器会将它们都读到同一个高速缓存行中，在多处理器下每个处理器都会缓存同样的头、尾节点，当一个处理器试图修改头节点时，会将整个缓存行锁定，那么在缓存一致性机制的作用下，会导致其他处理器不能访问自己高速缓存中的尾节点，而队列的入队和出队操作则需要不停地修改头节点和尾节点，所以在多处理器的情况下将会严重影响到队列的入队和出队效率。使用追加到 64 字节的方式来填满高速缓冲区的缓存行，避免头节点和尾节点加载到同一个缓存行，使头、尾节点在修改时不会互相锁定。

注意：P6 系统和奔腾处理器，由于 L1 和 L2 高速缓存行是 32 个字节宽，所以不适用于该方式。

此外，如果共享变量不被频繁写的话，锁的几率也非常小，所以就没必要通过追加字节的方式来避免相互锁定，因为使用追加字节的方式需要处理器读取更多的字节到高速缓冲区，本身就会带来一定的性能消耗。

总的来说，`LinkedTransferQueue` 的性能分别是 `SynchronousQueue` 的 3 倍（非公平模式）和 14 倍（公平模式）。因为像 `ThreadPoolExecutor` 这样的类在任务传递时都是使用 `SynchronousQueue`，所以使用 `LinkedTransferQueue` 来代替 `SynchronousQueue` 也会使得 `ThreadPoolExecutor` 得到相应的性能提升。考虑到 `executor` 在并发编程中的重要性，你就会理解添加这个实现类的重要性了。Java5 中的 `SynchronousQueue` 使用两个队列（一个用于正在等待的生产者、另一个用于正在等待的消费者）和一个用来保护两个队列的锁。而 `LinkedTransferQueue` 使用 CAS 操作（译者注：参考 wiki）实现一个非阻塞的方法，这是避免序列化处理任务的关键。⁶

5.4.5.7 LinkedBlockingDeque

JDK6 提供了一种双段队列(Doubled-Ended Queue)，简称 Deque。Deque 允许在队列的头部或者尾部进行出队和入队操作。与 Queue 相比，它们具有更加复杂的功能。

`LinkedList`、`ArrayDeque` 和 `LinkedBlockingDeque`，都实现了双端队列 Deque 接口。其中 `LinkedList` 使用链表实现了双端队列，`ArrayDeque` 使用数组实现双端队列。通常情况下，由于 `ArrayDeque` 基于数组实现，拥有高效的随机访问性能，因此 `ArrayDeque` 具有更好的遍历性能。但是当队列的大小变化较大时，`ArrayDeque` 需要重新分配内存，并进行数组复制，在这种环境下，基于链表的 `LinkedList` 没有内存调整和数据复制的负担，性能表现较好。但无论是 `LinkedList` 或者 `ArrayDeque`，它们都不是线程安全的。`LinkedBlockingDeque` 是一个线程安全的双端队列实现，可以说，它已经是最为复杂的一个队列实现。在内部实现中，`LinkedBlockingDeque` 使用链表结构。每一个队列节点都维护一个前驱结点和一个后驱节点。`LinkedBlockingDeque` 没有进行读写锁的分离，因此同一时间只能有一个线程对其进行操作。因此，在高并发应用中，它的性能表现要远远低于 `LinkedBlockingQueue`，更要低于 `ConcurrentLinkedQueue`。

`ArrayDeque` 继承了 Deque（双向队列）接口，使用此类可以自己实现 `java.util.Stack` 类的功能，

⁶ 来源于 William Scherer, Doug Lea, and Michael Scott 编写的论文：<http://www.cs.rice.edu/~wnsl/papers/2006-PPoPP-SQ.pdf>

去掉了 `java.util.Stack` 的多线程同步的功能。

`java.util.ArrayDeque` 的源码如下所示。

代码清单 5-75 ArrayDeque 源代码

```
private transient E[] elements;
private transient int head;
private transient int tail;
/*此处存放e的位置是从elements数组最后的位置开始存储的*/
public void addFirst(E e) {
    if (e == null)
        throw new NullPointerException();
    elements[head = (head - 1) & (elements.length - 1)] = e; //首次数组容量默认为16, head=(0-1) & (16-1)=15
    if (head == tail)
        doubleCapacity();
}
/*每次扩容都按插入的先后顺序重新放入一个新的数组中, 最新插入的放在数组的第一个位置。*/
private void doubleCapacity() {
    assert head == tail;
    int p = head;
    int n = elements.length;
    int r = n - p; // number of elements to the right of p
    int newCapacity = n << 1;
    if (newCapacity < 0)
        throw new IllegalStateException("Sorry, deque too big");
    Object[] a = new Object[newCapacity];
    System.arraycopy(elements, p, a, 0, r);
    System.arraycopy(elements, 0, a, r, p);
    elements = (E[])a;
    head = 0;
    tail = n;
}
public E removeFirst() {
    E x = pollFirst();
    if (x == null)
        throw new NoSuchElementException();
    return x;
}
public E pollFirst() {
    int h = head;
    E result = elements[h]; // Element is null if deque empty
    if (result == null)
        return null;
    elements[h] = null; // 重新设置数组中的这个位置为null, 方便垃圾回收。
    head = (h + 1) & (elements.length - 1); //将head的值回退, 相当于将栈的指针又向下移动一格。例如, 12-->13
    return result;
}
```

```
}  
public E peekFirst() {  
    return elements[head]; // elements[head] is null if deque empty  
}
```

如果我们希望创建一个存放 `Integer` 类型的 `Stack`，只要在类中创建一个 `ArrayDeque` 类的变量作为属性，之后定义的出栈、入栈，观察栈顶元素的操作就直接操作 `ArrayDeque` 的实例变量即可。

代码清单 5-76 Integer 类型的 Stack

```
import java.util.ArrayDeque;  
import java.util.Deque;  
public class IntegerStack {  
    private Deque<Integer> data = new ArrayDeque<Integer>();  
    public void push(Integer element) {  
        data.addFirst(element);  
    }  
    public Integer pop() {  
        return data.removeFirst();  
    }  
    public Integer peek() {  
        return data.peekFirst();  
    }  
    public String toString()  
        return data.toString();  
    }  
    public static void main(String[] args) {  
        IntegerStack stack = new IntegerStack();  
        for (int i = 0; i < 5; i++) {  
            stack.push(i);  
        }  
        System.out.println(stack);  
        System.out.println("After pushing 5 elements: " + stack);  
        int m = stack.pop();  
        System.out.println("Popped element = " + m);  
        System.out.println("After popping 1 element : " + stack);  
        int n = stack.peek();  
        System.out.println("Peeked element = " + n);  
        System.out.println("After peeking 1 element : " + stack);  
    }  
}
```

- (1) `ArrayDeque` 有两个类属性，`head` 和 `tail`，两个指针。
- (2) `ArrayDeque` 通过一个数组作为载体，其中的数组元素在 `add` 等方法执行时不移动，发生变化的只是 `head` 和 `tail` 指针，而且指针是循环变化，数组容量不限制。
- (3) `offer` 方法和 `add` 方法都是通过其中的 `addLast` 方法实现，每添加一个元素，就把元素加到数组的尾部，此时，`head` 指针没有变化，而 `tail` 指针加一，因为指针是循环加的，所

以当 tail 追上 head ((this.tail = this.tail + 1 & this.elements.length - 1) == this.head) 时, 数组容量翻一倍, 继续执行。

(4) remove 方法和 poll 方法都是通过其中的 pollFirst 方法实现, 每移除一个元素, 该元素所在位置变成 null, 此时, tail 指针没有变化, 而 head 指针加一, 当数组中没有数据时, 返回 null。

(5) 因为 ArrayDeque 不是线程安全的, 所以, 用作堆栈时快于 Stack, 在用作队列时快于 LinkedList。

5.4.5 并发工具类

在 JDK 的开发包里提供了几个非常有用的并发工具类。CountDownLatch、CyclicBarrier 和 Semaphore 工具类提供了一种并发流程控制的手段, CountDownLatch 是能使一组线程等另一组线程都跑完了再继续跑; CyclicBarrier 能够使一组线程在一个时间点上达到同步, 可以是一起开始执行全部任务或者一部分任务。同时, 它是可以循环使用的; Semaphore 是只允许一定数量的线程同时执行一部分任务。Exchanger 工具类则提供了在线程间交换数据的一种手段。

5.4.5.1 CountDownLatch 工具类

直译过来就是倒计时(CountDown)门闩(Latch)。倒计时不用说, 门闩的意思顾名思义就是阻止前进。在这里就是指 CountDownLatch.await()方法在倒计数为 0 之前会阻塞当前线程。

CountDownLatch 的作用和 Thread.join()方法类似, 可用于一组线程和另外一组线程的协作。例如, 主线程在做一项工作之前需要一系列的准备工作, 只有这些准备工作都完成, 主线程才能继续它的工作。这些准备工作彼此独立, 所以可以并发执行以提高速度。在这个场景下就可以使用 CountDownLatch 协调线程之间的调度了。在直接创建线程的年代 (Java 5.0 之前), 我们可以使用 Thread.join()。在 JUC 出现后, 因为线程池中的线程不能直接被引用, 所以就必须使用 CountDownLatch 了。

CountDownLatch 允许一个或多个线程等待其他线程完成操作。

假设有这么一个场景: 我们需要解析一个 Excel 里多个 Sheet 的数据, 此时可以考虑使用多线程, 每个线程解析一个 Sheet 里的数据, 等到所有的 Sheet 都解析完之后, 程序需要提示解析完成。在这个需求中, 要实现主线程等待所有线程完成 Sheet 的解析操作, 最简单的做法是使用 join()方法, 如以下代码所示。

代码清单 5-77 join 示例

```
public class JoinCountDownLatchTest {
    public static void main(String[] args) throws InterruptedException{
        Thread parser1 = new Thread(new Runnable(){

            @Override
            public void run() {
                // TODO Auto-generated method stub
                System.out.println("parser1 finish!");
            }
        });
    }
}
```

```

    });

    Thread parser2 = new Thread(new Runnable() {
        @Override
        public void run() {
            // TODO Auto-generated method stub
            System.out.println("parser2 finish!");
        }
    });

    parser1.start();
    parser2.start();
    parser1.join();
    parser2.join();
    System.out.println("all parser finish");
}
}

```

join()方法用于让当前执行线程等待 join 线程执行结束。其实现原理是不停检查 join 线程是否存活, 如果 join 线程存活则让当前线程永远等待。直到 join 线程中之后, 线程的 this.notifyAll()方法会被调用, 调用 notifyAll()方法是在 JVM 里实现的。

在 JDK5 之后的并发包中提供的 CountDownLatch 也可以实现 join 的功能。

CountDownLatch 的构造函数接收一个 int 类型的参数作为计数器, 如果你想等待 N 个工作完成, 这里就传入 N 。当我们调用 CountDownLatch 的 countDown 方法时, N 就会减小 1, CountDownLatch 的 await 方法会阻塞当前线程, 直到 N 变成零。由于 countDown 方法可以用在任何地方, 所以这里说的 N 个点, 可以是 N 个线程, 也可以是 1 个线程里的 N 个执行步骤。用在多个线程时, 只需要把这个 CountDownLatch 的引用传递到线程里即可。

CountDownLatch 类是一个同步计数器, 构造时传入 int 参数, 该参数就是计数器的初始值, 每调用一次 countDown()方法, 计数器减 1, 计数器大于 0 时, await()方法会阻塞程序继续执行 CountDownLatch 如其所写, 是一个倒计数的锁存器, 当计数减至 0 时触发特定的事件。利用这种特性, 可以让主线程等待子线程的结束。下面以一个模拟运动员比赛的例子加以说明。

代码清单 5-78 CountDownLatchDemo 类代码

```

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.Executor;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class CountDownLatchDemo {
    private static final int PLAYER_AMOUNT = 5;
    public CountDownLatchDemo() {

```



```

    // TODO Auto-generated constructor stub
}
/**
 * @param args
 */
public static void main(String[] args) {
    // TODO Auto-generated method stub
    //对于每位运动员, CountdownLatch 减 1 后即结束比赛
    CountdownLatch begin = new CountdownLatch(1);
    //对于整个比赛, 所有运动员结束后才算结束
    CountdownLatch end = new CountdownLatch(PPLAYER_AMOUNT);
    Player[] plays = new Player(PPLAYER_AMOUNT);
    for(int i=0;i<PPLAYER_AMOUNT;i++)
        plays[i] = new Player(i+1,begin,end);
    //设置特定的线程池, 大小为 5
    ExecutorService exe = Executors.newFixedThreadPool(PPLAYER_AMOUNT);
    for(Player p:plays){
        exe.execute(p);
        //分配线程
        System.out.println("Race begins!");
        begin.countDown();
        try{
            end.await();
            //等待 end 状态变为 0, 即为比赛结束
        }catch (InterruptedException e) {
            // TODO: handle exception
            e.printStackTrace();
        }finally{
            System.out.println("Race ends!");
        }
        exe.shutdown();
    }
}
}

```

代码清单 5-79 Player 类代码

```

import java.util.concurrent.CountDownLatch;

public class Player implements Runnable {
    private int id;
    private CountdownLatch begin;
    private CountdownLatch end;
    public Player(int i, CountdownLatch begin, CountdownLatch end) {
        // TODO Auto-generated constructor stub
        super();
        this.id = i;
        this.begin = begin;
        this.end = end;
    }
    @Override

```

```

public void run() {
    // TODO Auto-generated method stub
    try{
        begin.await();
        //等待 begin 的状态为 0
        Thread.sleep((long)(Math.random()*100));
        //随机分配时间,即运动员完成时间
        System.out.println("Play"+id+" arrived.");
    }catch (InterruptedException e) {
        // TODO: handle exception
        e.printStackTrace();
    }finally{
        end.countDown();
        //使 end 状态减 1,最终减至 0
    }
}
}

```

总的来说,一句话可以总结 `CountDownLatch` 工具类, `CountDownLatch` 是一种 `Synchronizer`, 它可以延迟线程的进度直到线程的进度到线程到达终止状态。

5.4.5.2 `CyclicBarrier` 工具类

`CyclicBarrier` 翻译过来叫循环栅栏、循环障碍。它主要的方法就是一个: `await()`。`await()`方法每被调用一次,计数便会减少 1,并阻塞住当前线程。当计数减至 0 时,阻塞解除,所有在此 `CyclicBarrier` 上面阻塞的线程开始运行。在这之后,如果再次调用 `await()`方法,计数就又会变成 $N-1$,新一轮重新开始,这便是 `Cyclic` 的含义所在。

`CyclicBarrier` 的使用并不难,但需要注意它所相关的异常。除了常见的异常, `CyclicBarrier.await()` 方法会抛出一个独有的 `BrokenBarrierException`。这个异常发生在当某个线程在等待本 `CyclicBarrier` 时被中断或超时或被重置时,其他同样在这个 `CyclicBarrier` 上等待的线程便会受到 `BrokenBarrierException`。意思就是说,同志们,别等了,有个小伙伴已经挂了,咱们如果继续等有可能会一直等下去,所以各回各家吧。

`CyclicBarrier.await()`方法带有返回值,用来表示当前线程是第几个到达这个 `Barrier` 的线程。

和 `CountDownLatch` 一样, `CyclicBarrier` 同样可以可以在构造函数中设定总计数值。与 `CountDownLatch` 不同的是, `CyclicBarrier` 的构造函数还可以接受一个 `Runnable`,会在 `CyclicBarrier` 被释放时执行。

`CyclicBarrier` 工具类允许一组线程互相等待,直到达到某个公共屏障点(common barrier point)。在涉及一组固定大小的线程的程序中,这些线程必须不时地互相等待,此时 `CyclicBarrier` 很有用。因为该 barrier 在释放等待线程后可以重用,所以称它为循环的 barrier。`CyclicBarrier` 支持一个可选的 `Runnable` 命令,在一组线程中的最后一个线程到达之后(但在释放所有线程之前),该命令只在每个屏障点运行一次。若在继续所有参与线程之前更新共享状态,此屏障操作很有用。

在某种需求中,比如一个大型的任务,常常需要分配好多子任务去执行,只有当所有子任务都执行完成时候,才能执行主任务,这时候就可以选择 `CyclicBarrier` 了。

常用的方法是 `await` 方法，

```
public int await() throws InterruptedException, BrokenBarrierException
```

在所有参与者都已经在此 `barrier` 上调用 `await` 方法之前，将一直等待。如果当前线程不是将到达的最后一个线程，出于调度目的，将禁用它，且在发生以下情况之一前，该线程将一直处于休眠状态。

- 最后一个线程到达；或者
- 其他某个线程中断当前线程；或者
- 其他某个线程中断另一个等待线程；或者
- 其他某个线程在等待 `barrier` 时超时；或者
- 其他某个线程在此 `barrier` 上调用 `reset()`。

如果当前线程：

- 在进入此方法时已经设置了该线程的中断状态；或者
- 在等待时被中断。

则抛出 `InterruptedException`，并且清除当前线程的已中断状态。如果在线程处于等待状态时 `barrier` 被 `reset()`，或者在调用 `await` 时 `barrier` 被损坏，抑或任意一个线程正处于等待状态，则抛出 `BrokenBarrierException` 异常。

如果任何线程在等待时被中断，则其他所有等待线程都将抛出 `BrokenBarrierException` 异常，并将 `barrier` 置于损坏状态。

如果当前线程是最后一个将要到达的线程，并且构造方法中提供了一个非空的屏障操作，则在允许其他线程继续运行之前，当前线程将运行该操作。如果在执行屏障操作过程中发生异常，则该异常将传播到当前线程中，并将 `barrier` 置于损坏状态。

返回：

到达的当前线程索引，其中，索引 `getParties()-1` 指示将到达的第一个线程，零指示最后一个到达的线程。

抛出：

InterruptedException: 如果当前线程在等待时被中断

BrokenBarrierException: 如果另一个线程在当前线程等待时被中断或超时，或者重置了 `barrier`，或者在调用 `await` 时 `barrier` 被损坏，抑或由于异常而导致屏障操作（如果存在）失败。

示例代码如下所示。

代码清单 5-80 CyclicBarrierDemo

```
import java.io.IOException;
import java.util.Random;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
```

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class CyclicBarrierDemo {
    public static void main(String[] args) throws IOException, InterruptedException {
        //如果将参数改为 4, 但是下面只加入了 3 个选手, 这永远等待下去
        //Waits until all parties have invoked await on this barrier.
        CyclicBarrier barrier = new CyclicBarrier(3);
        ExecutorService executor = Executors.newFixedThreadPool(3);
        executor.submit(new Thread(new Runner(barrier, "1 号选手")));
        executor.submit(new Thread(new Runner(barrier, "2 号选手")));
        executor.submit(new Thread(new Runner(barrier, "3 号选手")));
        executor.shutdown();
    }
}

```

```

class Runner implements Runnable {
    // 一个同步辅助类, 它允许一组线程互相等待, 直到到达某个公共屏障点 (common barrier point) 19.
    private CyclicBarrier barrier;
    private String name;
    public Runner(CyclicBarrier barrier, String name) {
        super();
        this.barrier = barrier;
        this.name = name;
    }
    @Override
    public void run() {
        try {
            Thread.sleep(1000 * (new Random()).nextInt(8));
            System.out.println(name + " 准备好了...");
            // barrier 的 await 方法, 在所有参与者都已经在此 barrier 上调用 await 方法之前,
            将一直等待。
            barrier.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
        System.out.println(name + " 起跑!");
    }
}

```

运行输出如代码清单 5-81 所示。

代码清单 5-81 CyclicBarrierDemo

```

1 号选手 准备好了...
3 号选手 准备好了...
2 号选手 准备好了...

```


2号选手 起跑!

1号选手 起跑!

3号选手 起跑!

除了前面介绍的 await 方法以外, 还有以下方法。

await(): 在所有参与者都已经在此 barrier 上调用 await 方法之前, 将一直等待。

await(long timeout, TimeUnit unit): 在所有参与者都已经在此屏障上调用 await 方法之前, 将一直等待。

getNumberWaiting(): 返回当前在屏障处等待的参与者数目。

getParties(): 返回要求启动此 barrier 的参与者数目。

isBroken(): 查询此屏障是否处于损坏状态。

reset(): 将屏障重置为其初始状态。

如果在构造 CyclicBarrier 对象的时候传了一个 Runnable 对象进去, 则每次到达公共屏障点的时候都最先执行这个传进去的 Runnable, 然后再执行处于等待的 Runnable。

代码清单 5-82 CyclicBarrierDemo1

```
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class CyclicBarrierDemo1 {
    public static void main(String[] args) {
        ExecutorService service = Executors.newCachedThreadPool();
        //final CyclicBarrier cb = new CyclicBarrier(3); //创建 CyclicBarrier 对象并
        设置 3 个公共屏障点
        final CyclicBarrier cb = new CyclicBarrier(3, new Runnable() {
            @Override
            public void run() {
                System.out.println("*****我最先执行*****");
            }
        });
        for (int i = 0; i < 3; i++) {
            Runnable runnable = new Runnable() {
                public void run() {
                    try {
                        Thread.sleep((long) (Math.random() * 10000));
                        System.out.println("线程" + Thread.currentThread().getName()
+
                            "即将到达集合地点 1, 当前已有" + cb.getNumberWaiting() + "个已
经到达, 正在等候");
                        cb.await(); //到此如果没有达到公共屏障点, 则该线程处于等待状态, 如果
                        达到公共屏障点则所有处于等待的线程都继续往下运行
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            };
            service.execute(runnable);
        }
    }
}
```

```

        Thread.sleep((long) (Math.random() * 10000));
        System.out.println("线程" + Thread.currentThread().getName() +
            "即将到达集合地点 2, 当前已有" + cb.getNumberWaiting() + "个已
经到达, 正在等候");
        cb.await(); //这里 CyclicBarrier 对象又可以重用
        Thread.sleep((long) (Math.random() * 10000));
        System.out.println("线程" + Thread.currentThread().getName() +
            "即将到达集合地点 3, 当前已有" + cb.getNumberWaiting() + "个已
经到达, 正在等候");
        cb.await();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

};
service.execute(runnable);
}
service.shutdown();
}
}

```

运行输出如代码清单 5-83 所示。

代码清单 5-83 CyclicBarrierDemo1

```

线程 pool-1-thread-1 即将到达集合地点 1, 当前已有 0 个已经到达, 正在等候
线程 pool-1-thread-3 即将到达集合地点 1, 当前已有 1 个已经到达, 正在等候
线程 pool-1-thread-2 即将到达集合地点 1, 当前已有 2 个已经到达, 正在等候
*****我最先执行*****
线程 pool-1-thread-2 即将到达集合地点 2, 当前已有 0 个已经到达, 正在等候
线程 pool-1-thread-1 即将到达集合地点 2, 当前已有 1 个已经到达, 正在等候
线程 pool-1-thread-3 即将到达集合地点 2, 当前已有 2 个已经到达, 正在等候
*****我最先执行*****
线程 pool-1-thread-1 即将到达集合地点 3, 当前已有 0 个已经到达, 正在等候
线程 pool-1-thread-2 即将到达集合地点 3, 当前已有 1 个已经到达, 正在等候
线程 pool-1-thread-3 即将到达集合地点 3, 当前已有 2 个已经到达, 正在等候
*****我最先执行*****

```

CyclicBarrier 的功能也可以由 CountdownLatch 来实现。但是两者是有一些区别的, CountdownLatch 适用于一组线程和另一个主线程之间的工作协作。一个主线程等待一组工作线程的任务完毕才继续它的执行是使用 CountdownLatch 的主要场景; CyclicBarrier 用于一组或几组线程, 比如一组线程需要在一个时间点上达成一致, 例如同时开始一个工作。另外, CyclicBarrier 的循环特性和构造函数所接受的 Runnable 参数也是 CountdownLatch 所不具备的。

5.4.5.3 Semaphore 工具类

直译是信号量, 可能称它是许可量更容易理解。当然, 因为在计算机科学中这个名字由来已久, 所以不能乱改。它的功能比较好理解, 就是通过构造函数设定一个数量的许可, 然后通过 acquire 方法获得许可, release 方法释放许可。它还有 tryAcquire 和 acquireUninterruptibly 方法, 可以根据

自己的需要选择。

`public void acquire() throws InterruptedException` 从此信号量获取一个许可，在提供一个许可前一直将线程阻塞，否则线程被中断。获取一个许可（如果提供了一个）并立即返回，将可用的许可数减 1。

如果没有可用的许可，则在发生以下两种任意情况前，禁止将当前线程用于线程安排目的并使其处于休眠状态。

- 某些其他线程调用此信号量的 `release()` 方法，并且当前线程是下一个要被分配许可的线程。
- 其他某些线程中断当前线程。

如果当前线程：

- 被此方法将其已中断状态设置为 `on`。
- 在等待许可时被中断。

则抛出 `InterruptedException`，并且清除当前线程的已中断状态。

抛出：`InterruptedException` - 如果当前线程被中断。

`public void release()` 方法：释放一个许可，将其返回给信号量。释放一个许可，将可用的许可数增加 1。如果任意线程试图获取许可，则选中一个线程并将刚刚释放的许可给予它。然后针对线程安排目的启用（或再启用）该线程。不要求释放许可的线程必须通过调用 `acquire()` 来获取许可。通过应用程序中的编程约定来建立信号量的正确用法。

代码清单 5-84 SemaphoreDemo

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

public class SemaphoreDemo {
    public static void main(String[] args) {
        // 线程池
        ExecutorService exec = Executors.newCachedThreadPool();
        // 只能 5 个线程同时访问
        final Semaphore semp = new Semaphore(5);
        // 模拟 20 个客户端访问
        for (int index = 0; index < 20; index++) {
            final int NO = index;
            Runnable run = new Runnable() {
                public void run() {
                    try {
                        // 获取许可
                        semp.acquire();
                        System.out.println("Accessing: " + NO);
                        Thread.sleep((long) (Math.random() * 10000));
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            };
            exec.execute(run);
        }
    }
}
```

```

// 访问完后,释放,如果屏蔽下面的语句,则在控制台只能打印 5 条记录,
之后线程一直阻塞
        semp.release();
    } catch (InterruptedException e) {
    }
}
};
exec.execute(run);
}
// 退出线程池
exec.shutdown();
}
}

```

运行输出如清单 5-85 所示。

代码清单 5-85 SemaphoreDemo

```

Accessing: 0
Accessing: 2
Accessing: 4
Accessing: 1
Accessing: 6
Accessing: 8
Accessing: 10
Accessing: 12
Accessing: 17
Accessing: 3
Accessing: 5
Accessing: 7

```

上面的例子只允许 5 个线程同时进入执行 `acquire()` 和 `release()` 之间的代码。

5.4.5.4 Exchanger 工具类

`Exchanger` 可以在两个线程之间交换数据,只能是 2 个线程,他不支持更多的线程之间互换数据。

当线程 A 调用 `Exchange` 对象的 `exchange()` 方法后,他会陷入阻塞状态,直到线程 B 也调用了 `exchange()` 方法,然后以线程安全的方式交换数据,之后线程 A 和 B 继续运行。

代码清单 5-86 ThreadLocalDemo

```

import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.Exchanger;

public class ThreadLocalDemo {
    public static void main(String[] args) {

```



```

    Exchanger<List<Integer>> exchanger = new Exchanger<List<Integer>>();
    new Consumer(exchanger).start();
    new Producer(exchanger).start();
}

```

```

}

```

```

class Producer extends Thread {
    List<Integer> list = new ArrayList<Integer>();
    Exchanger<List<Integer>> exchanger = null;
    public Producer(Exchanger<List<Integer>> exchanger) {
        super();
        this.exchanger = exchanger;
    }
    @Override
    public void run() {
        Random rand = new Random();
        for(int i=0; i<10; i++) {
            list.clear();
            list.add(rand.nextInt(10000));
            list.add(rand.nextInt(10000));
            list.add(rand.nextInt(10000));
            list.add(rand.nextInt(10000));
            list.add(rand.nextInt(10000));
            try {
                list = exchanger.exchange(list);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}

```

```

class Consumer extends Thread {
    List<Integer> list = new ArrayList<Integer>();
    Exchanger<List<Integer>> exchanger = null;
    public Consumer(Exchanger<List<Integer>> exchanger) {
        super();
        this.exchanger = exchanger;
    }
    @Override
    public void run() {
        for(int i=0; i<10; i++) {
            try {
                list = exchanger.exchange(list);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block

```

```
e.printStackTrace();  
    }  
    System.out.print(list.get(0)+" ", "  
    System.out.print(list.get(1)+" ", "  
    System.out.print(list.get(2)+" ", "  
    System.out.print(list.get(3)+" ", "  
    System.out.println(list.get(4)+" ", "  
    }  
}  
}
```

运行输出如清单 5-87 所示。

代码清单 5-87 ThreadLocalDemo

```
6825, 4001, 7074, 7624, 6241,  
4657, 1334, 3052, 1267, 4737,  
4208, 3299, 2163, 8197, 2683,  
4207, 141, 5700, 1485, 771,  
6906, 5448, 2633, 563, 7218,  
433, 5361, 8148, 9710, 1737,  
267, 2610, 8424, 9874, 4910,  
5185, 7682, 2220, 7114, 7467,  
7823, 2242, 7842, 551, 626,  
2356, 9511, 4107, 7537, 688,
```

5.5 本章小结

本章首先针对并行程序优化的一些普遍型概念、技巧进行了介绍，包括进程、线程方面的概念性知识，也对 Synchronized、Volatile、锁、线程池等很实用的使用技巧进行了总结，接下来对增强程序并行能力的几个技巧进行了阐述，最后针对 JDK 自带的一些类库，例如并行容器、队列、工具类等进行了解释，特别是对这些类库的使用进行了详细的描述。

6 chapter

第6章 JVM 性能测试及监控

《三国志·魏志·华佗传》：府吏倪寻、李延共止，俱头痛身热，所苦正同。佗曰：“寻当下之，延当发汗。”或难其异，佗曰：“寻外实，延内实，故治之宜殊。”即各与药，明旦并起。

Java 程序设计优化不可避免地需要与 JVM 打交道，在提出优化方案之前，我们可以有效地利用 JVM 自带检测方式、各种开源工具来帮助我们找出性能瓶颈，最终帮助我们对症下药。

总的来说，改善性能需要 3 个步骤，即性能监控、性能分析、性能调优。

性能监控：一种以非强行或者入侵方式收集或查看应用运营性能数据的活动。监控通常是指一种在生产、质量评估或者开发环境下实施的带有预防或主动性的活动。当应用相关干系人提出性能问题却没有提供足够多的线索时，首先我们需要进行性能监控，随后是性能分析。

性能分析：一种以侵入方式收集运行性能数据的活动，它会影响应用的吞吐量或响应性。性能分析是针对性能问题的答复结果，关注的范围通常比性能监控更加集中。性能分析很少在生产环境下进行，通常是在质量评估、系统测试或者开发环境下进行，是性能监控之后的步骤。

性能调优：一种为改善应用响应性或吞吐量而更改参数、源代码、属性配置的活动，性能调优是在性能监控、性能分析之后的活动。

通过以上三步骤，我们可以定位、为解决问题找到依据，本章解决的是前两个步骤，第 3、4、5、7、8 章解决第三个步骤。

本章主要介绍和解决以下问题，这也是下一章节的预备知识：

- 如何监控计算机设备。
- 如何监控应用程序。
- 如何监控 JVM。

6.1 监控计算机设备层

系统的整体性能由许多因素决定，例如 CPU 利用率、CPU 队列长度、磁盘空间和 I/O、内存使用情况、网络流量等。对于实时性要求较高的系统而言，对系统关键性指标的有效监控和有效管理是保证系统高可用性的重要手段，因此，务必制定出明确的系统性能策略规划，并对这些性能指标进行有效的实时监控。如果关键性能指标严重偏离或者系统发生故障，应该采取有效手段来准确定位问题引发的原因，并通过调优系统配置或改进应用程序等手段来有效提高系统的可用性。

为了监控设备层各个子章节里面介绍的工具或者程序能够在同一个测试样本上进行测试及现象描述，我们这里编写一个简单的应用程序，该程序会启动若干线程，每个线程会动态调整对于 CPU、内存、磁盘、网络、操作系统内部锁等的占用情况，通过程序帮助我们在同一维度了解各个工具或者程序的使用方式及优缺点。

代码清单 6-1 应用程序代码

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Random;
import java.util.Vector;

public class demoCode {
    public static class HoldCPUTask implements Runnable{
        public static Object[] lock = new Object[100];
        public static Random r = new Random();
        static{
            for(int i=0;i<lock.length;i++){
                lock[i] = new Object();
            }
        }

        @Override
        public void run() {
            int loop=0;

            // TODO Auto-generated method stub
            while(true){
                //随机占用 CPU 资源
                int loopNum=(int) (Math.random()*100); //产生一个 1-100 的随机数，该数值
                被用来确定冒泡算法基础数据个数
                int a[] = new int[loopNum];
                for(int i=0;i<loopNum;i++){
                    a[i] = (int) (Math.random()*100); //随机赋值
                }
            }
        }
    }
}
```



```
//开始冒泡方式排序
for(int i=1;i<loopNum;i++)
{
    for(int j=0;j<loopNum-i;j++)
    {
        if(a[j]>a[j+1])//比较交换相邻元素
        {
            int temp;
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
        }
    }
}
```

//随机占用磁盘 I/O

int fileloop=(int) (Math.random()*10000); //产生一个 1-10000 的随机数,
该数值被用来确定写入文件的次数

```
try{
    FileOutputStream fos = new FileOutputStream(new File("temp"));
    for(int i=0;i<fileloop;i++){
        fos.write(i);
    }
    fos.close();
    FileInputStream fis = new FileInputStream(new File("temp"));
    while(fis.read()!=-1);
}catch(FileNotFoundException e){
    e.printStackTrace();
}catch(IOException e){
    e.printStackTrace();
}
```

//随机开始持有锁

int x=(int) (Math.random()*100); //产生一个 1-100 的随机数, 该数值被用来确
定锁数量

```
synchronized(lock[x]){
    if(x%2==0){
        try {
            lock[x].wait(r.nextInt(10));
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } //等待
    }else{
        lock[x].notifyAll(); //通知
    }
}
```

```

        //随机开始占用内存
        int memSize=(int) (Math.random()*100); //产生一个 1-100 的随机数, 该数值
        被用来确定写入申请内存大小
        Vector v = new Vector();
        for(int i=0;i<=10;i++){
            byte[] b = new byte[memSize*memSize];
            v.add(b);
        }

        try {
            Thread.sleep(1000); //该线程休息一会
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

}

}

public static void main(String[] args){
    int threadNum=(int) (Math.random()*100); //产生一个 1-100 的随机数, 该数值被用来
    确定工作线程数量
    System.out.println(threadNum);
    for(int i=0;i<threadNum;i++){
        new Thread(new HoldCPUtask()).start();
    }
}
}

```

6.1.1 监控 CPU

如果我们想要让应用程序的性能或者程序的扩展性达到最高, 必须充分利用分配给它的 CPU 周期¹。我们要解决如何在多处理器、多核、异构型态服务器²上运行的多线程应用, 以便有效利用 CPU 周期。此外, 要特别注意的是, 应用程序消耗很多 CPU 资源并不意味着性能或者扩展性这两个层面达到了最高峰值。要想找出应用程序如何使用 CPU 周期, 可以在操作系统上监控 CPU 使用率。

大多数操作系统的 CPU 使用率分为用户态 CPU 使用率和系统态 CPU 使用率两类。用户态 CPU 使用率是指执行应用程序代码的时间占总 CPU 时间的百分比。系统态 CPU 使用率高意味着共享资源存在竞争或者 I/O 设备之间存在大量的交互。既然原本用于执行操作系统内核调用的 CPU

¹ 又称机器周期, 机器内部各种操作大致可归属为对 CPU 内部的操作和对主存的操作两大类, 由于 CPU 内部操作速度较快, CPU 访问一次内存所花的时间较长, 因此用从内存读取一条指令字的最短时间来定义, 这个基准时间就是 CPU 周期 (机器周期)。一个指令周期常由若干 CPU 周期构成。

² 本书指使用了与 CPU 不一样的 GPU 的机器。

周期也可以用来执行应用程序，所以理想情况下，应用程序达到最高性能和扩展性时，它的系统态 CPU 使用率应该为 0%。虽然这是理论值，但是提高应用性能和扩展性的目标时我们可以指定为尽可能降低系统态 CPU 的使用率。

对于计算密集型应用程序来说，不仅要监控用户态和系统态 CPU 使用率，还需要进一步监控每时钟指令数 (Instructions Per Clock, IPC) 或每指令时钟周期 (Cycles Per Instruction, CPI) 等指标。这两个指标对于计算密集型应用来说很重要，因为现代操作系统自带的 CPU 使用率监控工具只能报告 CPU 使用率，而没有 CPU 执行指令占用 CPU 时钟周期的百分比。这意味着，即便 CPU 在等待内存中的数据，操作系统工具仍然会报告 CPU 繁忙。我们把这种情况被称为停滞，当 CPU 执行指令而所用的操作数据不在寄存器或者缓存中时，就会发生停滞。由于指令执行前必须等待数据从内存被装入 CPU 寄存器，所以一旦发生停滞，就会浪费时钟周期。CPU 停滞通常会等待好几百个时钟周期。因此提高计算密集型应用性能的策略就是减少停滞或者改善 CPU 高速缓存使用率，从而减少 CPU 在等待内存数据时浪费的时钟周期。

执行路径长度与 CPI 之间有微妙的差别，执行路径长度与应用程序的算法选择关系密切，而 CPI 与编译器生成更有效地代码有关。前者着眼于通过选择合理的算法生成最短的 CPU 指令序列，后者着眼于让编译器生成最高效的代码，减少每条 CPU 指令上消耗的 CPU 时钟周期数。我们假设一个场景，这样比较有利于说明两者的实际情况。假设执行某 CPU 指令(载入数据操作)导致了 CPU 高速缓存未命中。由于 CPU 高速缓存未命中，CPU 需要从内存，而不是缓存，读取这些数据，最终完成该指令可能要消耗数百个 CPU 时钟周期。如果在编译器生成的指令序列之前插入预先获取指令，提前将载入操作要访问的数据从内存读取到缓存，这个“额外”的预先获取指令将大大减少载入操作消耗的时钟周期数。载入指令执行时 CPU 可以直接从缓存中读取需要的数据。不过由于新增加了预先获取指令，程序的执行路径长度、CPU 指令数都会相应增加。因此，有可能出现执行路径变长，CPU 时钟周期利用却更高效的情况。当然，这种情况出现的机会不一定很多，也有可能情况会更加糟糕。

我们本小节的监控方式演示都会按照 Windows 篇和 Linux 篇，其中 Windows 针对的是 Windows7 操作系统，Linux 针对的是 CentosV6.5 操作系统。

6.1.1.1 CPU 使用率监控工具——Windows 篇

Windows 上最常用的 CPU 使用率监控工具是任务管理器和性能监视器(以下采用英文名 Perfmon)。这两个监控工具用不同颜色区分用户态 CPU 使用率和系统态 CPU 使用率。

任务管理器

Windows 任务管理器提供了有关计算机性能的信息，并显示了计算机上所运行的程序和进程的详细信息。如果连接到网络，那么还可以查看网络状态并迅速了解网络是如何工作的。

任务管理器的用户界面提供了文件、选项、查看、窗口、关机、帮助等六大菜单项，各菜单下属还有应用程序、进程、性能、联网、用户等五个标签页，窗口底部则是状态栏，从这里可以查看到当前系统的进程数、CPU 使用比率、更改的内存容量等数据，默认设置下系统每隔两秒钟对数据进行 1 次自动更新，也可以点击“查看→更新速度”菜单重新设置。在 Windows7 中使用 Ctrl+Shift+Esc 组合键调出，也可以用鼠标右键点击任务栏选择“任务管理器”，另外 Ctrl+Alt+Delete 组合键也可以出现，只不过还要回到锁定界面。

我们在 Windows7 环境下点击清单 6-1 所示程序，通过任务管理可以获取 CPU 使用状态，如图 6-1 所示。



图6-1 任务管理器截图

上图中，我们可以看到刚才启动的 Java 程序（图片中的 javaw.exe）一直占据着较高的 CPU 使用率。

具体查看性能一览，我们可以看到 CPU 使用率、内存使用空间等，从图 6-2 中可以看到 CPU 的使用率抖动较大，这是因为运行的程序本身的随机性（随机数字控制资源占用）较大。

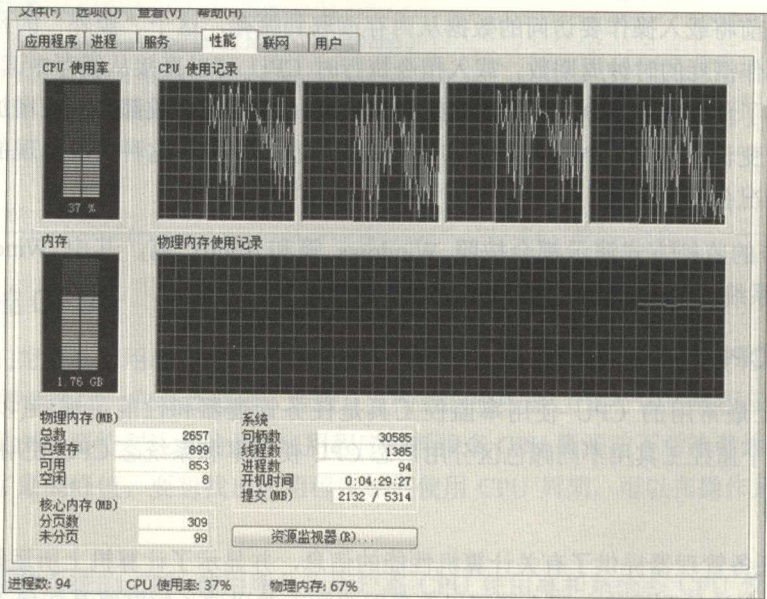


图6-2 任务管理器查看CPU使用

性能监视器（Perfmon）

Perfmon 提供了图表化的系统性能实时监视器、性能日志和警报管理，系统的性能日志可定义为二进制文件、文本文件、SQLSERVER 表记录等方式，可以很方便地使用第三方工具进行性能分析。Perfmon.exe 文件位于 C:\Windows\System32 目录下，Perfmon 工具可以通过在 cmd 命令环境下直接输入就能打开，如图 6-3 所示。

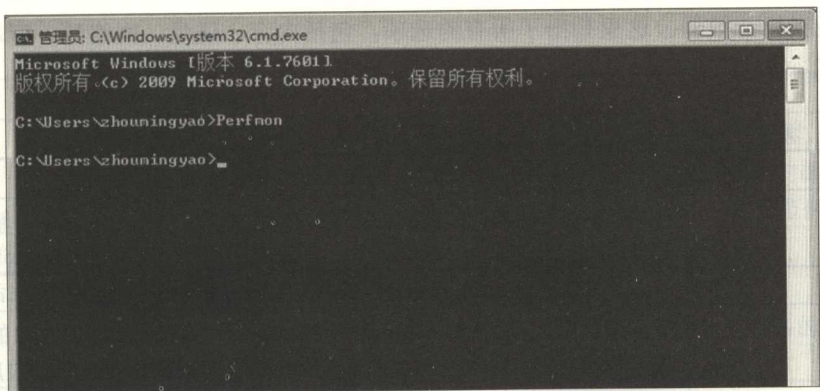


图6-3 启动Perfmon工具

打开的工具截图如图 6-4 所示。

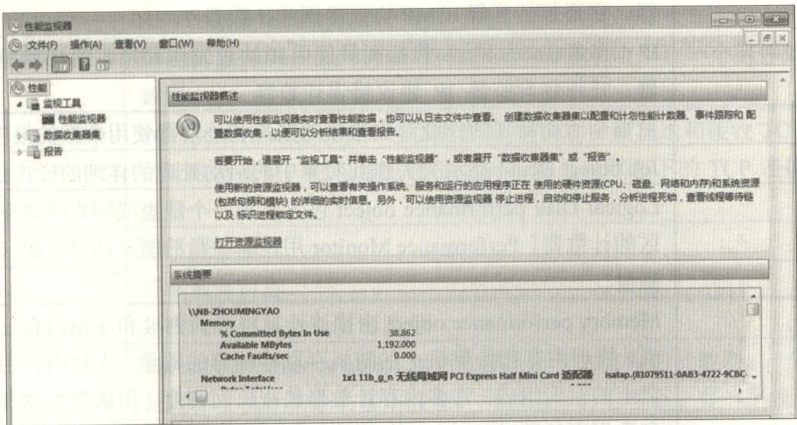


图6-4 Perfmon工具截图

点击性能监视器，然后点击添加按钮，可以添加我们需要的性能跟踪指标，如图 6-5 所示。

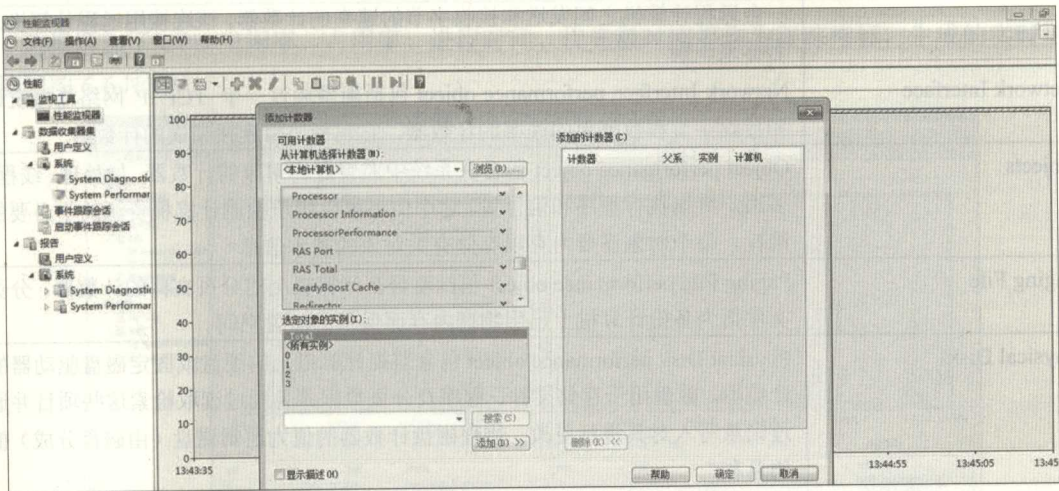


图6-5 Perfmon添加性能跟踪指标

Perfmon 提供了比较全面的系统性能指标，并且能够根据性能管理的要求定制日志内容、制定关键指标偏离时的警报措施。表 6-1 列出了 Perfmon 可以监控的性能对象，每一个性能对象项下

包含多个性能指标计数器。这里列出了所有的选项，包括内存、磁盘、网络等，所以下面各子章节就不再重复。

表 6-1 Perfmon 性能指标表

性能对象	对应信息
Browser	Browser performance object 由衡量通知、枚举和其他浏览器传输率的计数器组成
Cache	Cache performance object 包括监督文件系统缓存（物理内存上尽可能长时间地存储最近使用过的数据以便访问该数据时不需再从磁盘上读取的那一部分内存）的计数器。因为应用程序只使用缓存，因此该缓存可作为应用程序 I/O 操作的指示器。当有足够内存时，缓存可增大，但当内存不足时，缓存会变得太小而无法使用
ICMP	ICMP performance object 包括衡量用 ICMP 协议发送和接收消息的速度的计数器。它还包括监督 ICMP 协议错误的计数器
IP	IP performance object 包括衡量使用 IP 协议发送和接收的 IP 数据报速度的计数器。它还包含监督 IP 协议错误计数器
Job Object	由每个活动命名的作业对象收集的账户和处理器使用数据的报告
Job object Detail	Job object Detail 显示有关作业对象中的活动处理的详细的操作信息
Logical Disk	Logical Disk performance object 包含监视一个硬盘或固定磁盘驱动器的逻辑分区的计数器。Performance Monitor 用逻辑磁盘的驱动器号（如：C）来识别逻辑磁盘
Memory	Memory performance object 由描述计算机上的物理和虚拟内存行为的计数器组成。物理内存指计算机上的随机存取存储器的数量。虚拟内存由物理内存和磁盘上的空间组成。许多内存计数器监视页面调度（指磁盘与物理内存之间的代码和数据页的移动）。过多的页面调度（内存不足的一种表现）可引起拖延，会影响整个系统处理效率
NBT Connection	NBT Connection performance object 包括衡量用 NBT 连接在一台本地计算机和一台远程计算机之间发送和接收字节的速率的计数器。该连接用远程计算机的名称来识别
Network Interface	Network Interface performance object 包括衡量通过一个 TCP/IP 网络连接发送和接收字节和数据包的速率的计数器。它包括监督连接错误的计数器
Objects	Object performance object 包含在系统中监督逻辑对象的计数器，如处理、线程、多用户终端执行程序 and 信号量。这个信息可以用于检测计算机资源的不必要的消耗。每个对象需要内存以存储有关对象的基本信息
Paging File	Paging File performance object 包括监督在计算机上的分页文件的计数器。分页文件指为备份计算机上已用物理内存而保留的磁盘空间
Physical Disk	Physical Disk performance object 包含监视计算机上的硬盘或固定磁盘驱动器的计数器。磁盘用于存储文件、程序及分页数据并且通过读取检索这些项目并通过记录写入对其进行更改。物理磁盘计数器的值为逻辑磁盘（由磁盘分成）值的总和
Print Queue	显示一个打印列队的操作统计
Process	Process performance object 包含监视运行中应用程序和系统处理的计数器。所有在一个处理中的线程均共享同一个地址空间并可以访问同样的数据

续表

性能对象	对应信息
Processor	Processor performance object 包含衡量处理器活动方面的计数器。处理器是计算机进行算数和逻辑计算、在附属件起始操作及运行处理线程的部分。一台计算机可以有多台处理器。处理器对象将每台处理器作为对象的范例
Processor performance	处理器信息
PSched Pipe	数据包计划程序中的管道统计数
RAS Port	RAS Port performance object 包括监督计算机上的 RAS 设备的每个远程访问服务端口的计数器
RAS Total	RAS Total performance object 包含将计算机上的远程访问服务（RAS）设备的所有端口的值相加的计数器
Redirector	Redirector performance object 包括在本地计算机上监督网络连接的计数器
RSVP	RSVP 服务性能计数器
System	System performance object 包含应用于计算机上不止一个组件处理器范例的计数器
TCP	TCP performance object 包含衡量使用 TCP 协议发送和接收 TCP Segment 速率的计数器变量。它包含监督在每个 TCP 连接状态下的 TCP 连接数目的计数器变量
Telephony	电话服务系统
Terminal Services	终端服务信息
Terminal Services Session	每次终端服务会话资源监督
Thread	Thread performance object 包括衡量线程行为方面的计数器。一个线程是在一台处理器上执行指令的基本对象。所有运行的处理至少有一个线程
UDP	UDP performance object 包含衡量使用 UDP 协议发送和接收 UDP 数据报的速率的计数器。它包括监督 UDP 协议错误的计数器
WMI Objects	WMI 适配器返回的 WMI 高性能提供程序

这里添加了 CPU 空闲率、占用率、进程时间、优先级时间等作为跟踪指标，生成的实时更新图形如图 6-6 所示。

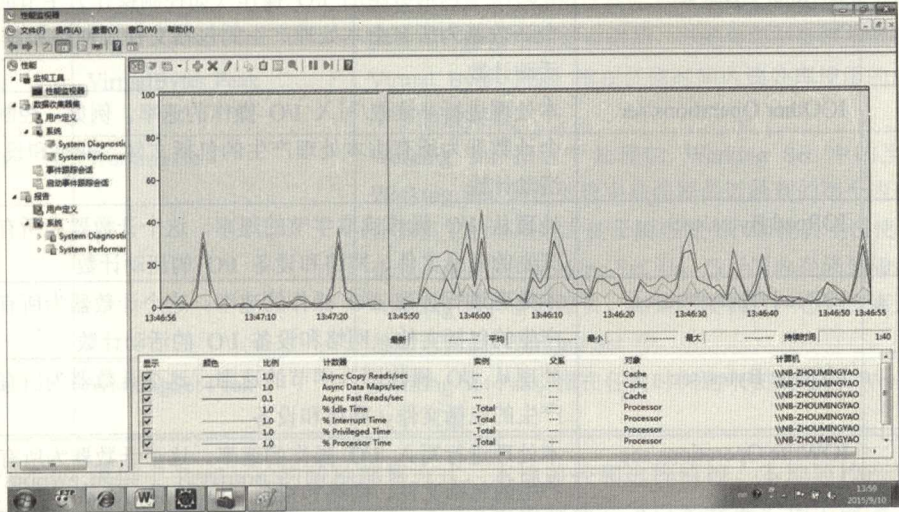


图6-6 Perfmon添加性能指标

以上列出的性能对象总共有上百个性能指标，我们关注一个系统的性能时，不可能关注这么多指标，有些对象对实际的应用系统影响并不大。这里只列出与 CPU 相关的指标，如表 6-2 所示。其他的指标请看对应的子章节。

表 6-2 Perfmon 的 CPU 指标

性能对象	计数器	提供的信息
Processor	% Idle Time	% Idle Time 是处理器在采样期间空闲的时间的百分比
Processor	%Processor Time	% Processor Time 指处理器用来执行非闲置线程时间的百分比。计算方法是，测量范例间隔内非闲置线程活动的时间，用范例间隔减去该值。这个计数器是处理器活动的主要说明器，显示在范例间隔时所观察的繁忙时间平均百分比
Processor	% User Time	% User Time 指处理器处于用户模式的时间百分比。用户模式是为应用程序、环境分系统和整数分系统设计的有限处理模式
Process	%Privileged Time	% Privileged Time 是在特权模式下处理线程执行代码所花时间的百分比。当调用 Windows 系统服务时，此服务经常在特权模式运行，以便获取对系统专有数据的访问。在用户模式执行的线程无法访问这些数据。对系统的调用可以是直接的（explicit）或间接的（implicit），例如页面错误或间隔
Process	Creating Process ID value	Creating Process ID value 指创建该进程的父进程号
Process	Elapsed Time	该进程运行的总时间（用秒计算）
Process	Handle Count	由这个处理现在打开的句柄总数。这个数字等于这个处理中每个线程当前打开的句柄的总数
Process	ID Process	ID Process 指这个处理的特别的识别符。ID Process 号可重复使用，所以这些 ID Process 号只能在一个处理的寿命期内识别那个处理
Process	IOData Bytes/sec	处理从 I/O 操作读取/写入字节的速率。这个计数器为所有由本处理产生的包括文件、网络和设备 I/O 的活动计数
Process	IOData Operations/sec	本处理进行读取/写入 I/O 操作的速率。这个计数器为所有由本处理产生的包括文件、网络和设备 I/O 的活动计数
Process	IOOther Bytes/sec	处理给不包括数据的 I/O 操作（如控制操作）字节的速率。这个计数器为所有由本处理产生的包括文件、网络和设备 I/O 的活动计数
Process	IOOther Operations/sec	本处理进行非读取/写入 I/O 操作的速率。例如，控制性能。这个计数器为所有由本处理产生的包括文件、网络和设备 I/O 的活动计数
Process	IORead Bytes/sec	处理从 I/O 操作读取字节的速率。这个计数器为所有由本处理产生的包括文件、网络和设备 I/O 的活动计数
Process	IORead Operations/sec	本处理进行读取 I/O 操作的速率。这个计数器为所有由本处理产生的包括文件、网络和设备 I/O 的活动计数
Process	IOWrite Bytes/sec	处理从 I/O 操作写入字节的速率。这个计数器为所有由本处理产生的包括文件、网络和设备
Process	IOWrite Operations/sec	本处理进行写入 I/O 操作的速率。这个计数器为所有由本处理产生的包括文件、网络和设备 I/O 的活动计数

续表

性能对象	计数器	提供的信息
Process	Page Faults/sec	Page Faults/sec 指在这个进程中执行线程造成的页面错误出现的速度。当线程引用了不在主内存工作集中的虚拟内存页即会出现 Page Fault。如果它在备用表中（即已经在主内存中）或另一个共享页的处理正在使用它，就会引起无法从磁盘中获取页
Process	Page File Bytes	Page File Bytes 指这个处理在 Paging file 中使用的最大字节数。Paging File 用于存储不包含在其他文件中的由处理使用的内存页。Paging File 由所有处理共享，并且 Paging File 空间不足会防止其他处理分配内存
Process	Page File Bytes Peak	Page File Bytes Peak 指这个处理在 Paging files 中使用的最大数量的字节
Process	Pool Nonpaged Bytes	Pool Nonpaged Bytes 指在非分页池中的字节数，非分页池是指系统内存（操作系统使用的物理内存）中可供对象（指那些在不处于使用时不可以写入磁盘上而且只要分派过就必须保留在物理内存中的对象）使用的一个区域。这个计数器仅显示上一次观察的值；而不是一个平均值
Process	PoolPaged Bytes	Pool Paged Bytes 指在分页池中的字节数，分页池是系统内存（操作系统使用的物理内存）中可供对象（在不处于使用时可以写入磁盘的）使用的一个区域。这个计数器仅显示上一次观察的值；而不是一个平均值
Process	Priority Base	这次处理的当前基本优先权。在一个处理中的线程可以根据处理的基本优先权提高或降低自己的基本优先权
Process	Private Bytes	Private Bytes 指这个处理不能与其他处理共享的、已分配的当前字节数
Process	Thread Count	在这次处理中正在活动的线程数目。指令是在一台处理器中基本的执行单位，线程是指执行指令的对象。每个运行处理至少有一个线程
Process	Virtual Bytes	Virtual Bytes 指处理使用的虚拟地址空间的以字节数显示的当前大小。使用虚拟地址空间不一定是指对磁盘或主内存页的相应的使用。虚拟空间是有限的，可能会限制处理加载数据库的能力
Process	VirtualBytes Peak	Virtual Bytes Peak 指在任何时间内该处理使用的虚拟地址空间字节的最大数
Process	Working Set	Working Set 指这个处理的 Working Set 中的当前字节数。Working Set 是在处理中被线程最近触到的那个内存页集。如果计算机上的可用内存处于阈值以上，即使页不在使用中，也会留在一个处理的 Working Set 中。当可用内存降到阈值以下，将从 Working Set 中删除页。如果需要页时，它会在离开主内存前软故障返回到 Working Set 中
Process	WorkingSet Peak	Working Set Peak 指在任何时间这个在处理的 Working Set 的最大字节数

Windows 提供了 Perfmon 的两种部署方式，本地监控和远程监控。本地监控产生的日志文件默认保存路径是 C:\perflogs 目录，在设置时可以根据需要在“日志文件”项下修改。本地监控产

生的日志文件除了可以在本机用性能监视器进行观测外，还可以外传到第三方监测分析平台上。远程监控可以实现对局域网内多台监控目标进行集中采样监控，其前提是监控主机与目标主机之间必须建立信任关系，并且打开相应的远程访问控制。在访问控制比较严格的环境下，远程监控难以部署。部署 Permon 时还应该考虑日志文件的存放问题，如果要长时间收集性能数据，最好调整一下采样间隔时间，如果采样间隔时间设置得太小，日志文件会快速增加。

Perfmon 的管理也有两种方法，控制台管理和命令行方式管理。可以通过运行 Perfmon.msc 调出性能管理的控制台，并根据监控策略制订、管理控制台。Perfmon 的另一种管理方式是命令行方式，Windows 提供了一个专门用于管理性能监控的命令——Logman，它不仅能够在命令行上启动和停止日志会话，还能够从命令行创建新的日志会话。

Typeperf

Windows 提供了一个显示当前性能指标的命令，Typeperf。Typeperf 能方便地采集到系统性能数据，但仅仅是获取数据，并不产生警报和日志记录的动作。用 Typeperf 可以得到前面提到的 Perfmon 的所有指标值。Typeperf 的标准输出是屏幕显示，我们可以通过输出重定义将结果输出到文本文件当中，并将结果文件传给第三方系统。Windows typeperf 是收集操作系统性能统计数据的命令行工具。typeperf 可以在 Windows 命令提示符窗口中运行，或者作为脚本语句在 bat 或 cmd 的文件中运行。这种应用方式下，Typeperf 的动作由第三方软件根据需要来管理，也可以通过定制计划任务来定时启动。

Typeperf 的使用方式如清单 6-2 所示。

代码清单 6-2 Typeperf 使用方式

```
Microsoft (R) TypePerf.exe (5.2.3790.0)
(C) Microsoft Corporation. All rights reserved.
Typeperf 将性能数据写入命令窗口或日志文件。要停止 Typeperf，请按 CTRL+C。
用法：
typeperf { <counter [counter ...]>
| -cf <filename>
| -q [object]
| -qx [object]
} [options]
参数：
<counter [counter ...]>      要监视的性能计数器。
选项：
-?                          显示跟上下文相关的帮助。
-f <CSV|TSV|BIN|SQL>        输出文件格式。默认值是 CSV。
-cf <filename>              含有监视的性能计数器的文件，一个计数器一行。
-si <[[hh:]mm:]ss>          示例间的时间。默认值是 1 秒。
-o <filename>               输出文件或 SQL 数据库的路径。默认值为 STDOUT。
-q [object]                 列出已安装的计数器(无实例)。要列出某个对象的计数器，包括对象
名，如 Processor。
-qx [object]                列出已安装的计数器(带实例)。要列出某个对象的计数器，包括对象
名，如 Processor。
-sc <samples>               要收集的示例数量。默认值为，在 CTRL+C 之前都进行采样。
```


-config <filename> 含有命令选项的设置文件。
 -s <computer_name> 在计数器路径中没有指定服务器的情况下要监视的服务器。
 -y 不用提示对所有问题都回答 yes。

注意:

Counter 是性能计数器的全名, 格式为

"//<Computer>/<Object>(<Instance>)/<Counter>";

例如 "//Server1/Processor(0)/% User Time"。

例如:

```
typeperf "/Processor(_Total)/% Processor Time"
```

```
typeperf -cf counters.txt -si 5 -sc 50 -f TSV -o domain2.tsv
```

```
typeperf -qx PhysicalDisk -o counters.txt
```

typeperf 也可以被用来监控运行队列长度。typeperf 接受性能计数器的名字作为参数, 然后以列表方式打印性能数据。性能计数器\System\Processor Queue Length 监控运行队列长度, 所以可以使用下列命令 typeperf "\System\Processor Queue Length" 监控运行队列长度。

6.1.1.2 CPU 使用率监控工具——Linux 篇

GNOME System Monitor

Linux 上可以使用图形化工具 GNOME System Monitor 监控 CPU 使用率。运行后界面如图 6-7 所示, 可以清楚地看到 CPU、内存、网络 I/O 等各项数据指标及历史数据。

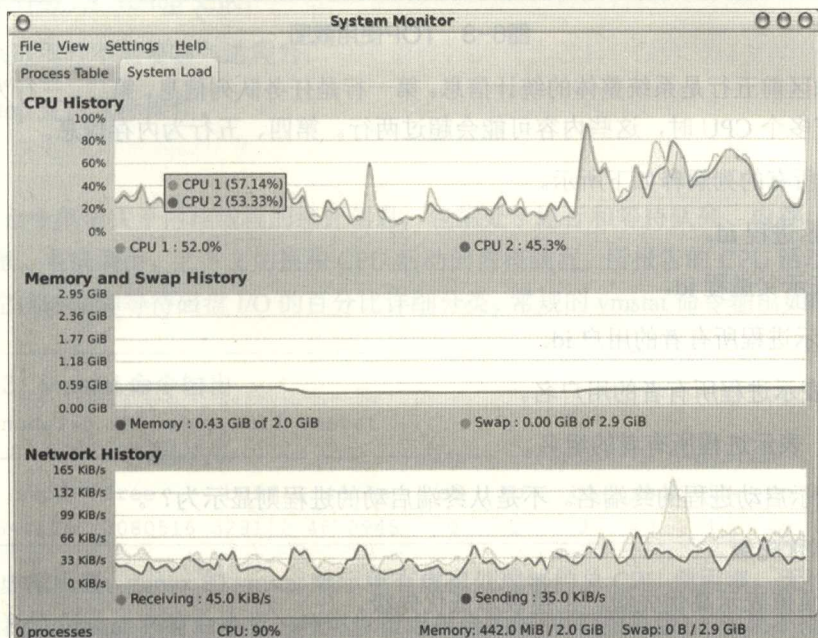


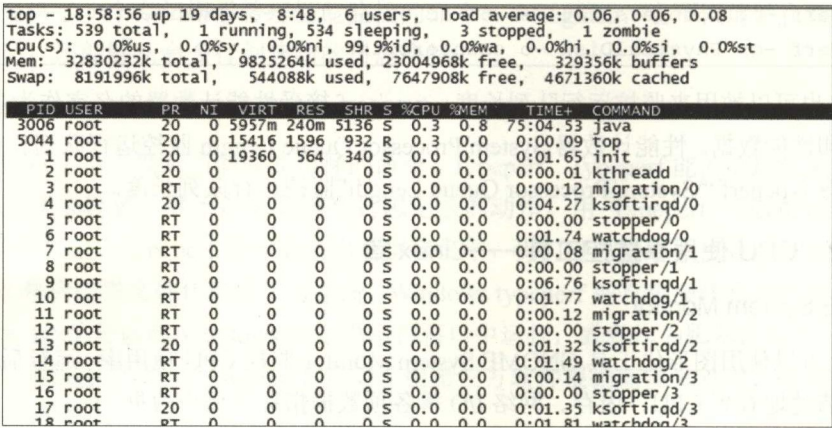
图6-7 GNOME使用

因为图形化界面采用的机会较少, Linux 下有大量的命令行共同存在, 所以这里不多做介绍。

前面说了, 除了图形化工具以外, Linux 也提供了监控 CPU 使用率的命令行工具, 可以通过保留文本形式的保留 CPU 使用率运行历史。这类型的工具很多, 例如 vmstat、top、pidstat 等, 下面将逐一介绍。

Top 命令

Top 命令不仅包括 CPU 使用率也包括进程统计数据 and 内存使用率。Top 是一个动态显示过程，即可以通过用户按键来不断刷新当前状态。如果在前台执行该命令，它将独占前台，直到用户终止该程序为止。比较准确地说，Top 命令提供了实时的对系统处理器的状态监视，它将显示系统中 CPU 最“敏感”的任务列表。该命令可以按 CPU 使用、内存使用和执行时间对任务进行排序；而且该命令的很多特性都可以通过交互式命令或者在个人定制文件中进行设定。Top 使用截图如图 6-8 所示。



```
top - 18:58:56 up 19 days, 8:48, 6 users, load average: 0.06, 0.06, 0.08
Tasks: 539 total, 1 running, 534 sleeping, 3 stopped, 1 zombie
Cpu(s): 0.0%us, 0.0%sy, 0.0%ni, 99.9%id, 0.0%wa, 0.0%hi, 0.0%st
Mem: 32830232k total, 9825264k used, 23004968k free, 329356k buffers
Swap: 8191996k total, 544088k used, 7647908k free, 4671360k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3006	root	20	0	5957m	240m	5136	S	0.3	0.8	75:04.53	java
5044	root	20	0	15416	1596	932	R	0.3	0.0	0:00.09	top
1	root	20	0	19360	564	340	S	0.0	0.0	0:01.65	init
2	root	20	0	0	0	0	S	0.0	0.0	0:00.01	kthreadd
3	root	RT	0	0	0	0	S	0.0	0.0	0:00.12	migration/0
4	root	20	0	0	0	0	S	0.0	0.0	0:04.27	ksoftirqd/0
5	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	stopper/0
6	root	RT	0	0	0	0	S	0.0	0.0	0:01.74	watchdog/0
7	root	RT	0	0	0	0	S	0.0	0.0	0:00.08	migration/1
8	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	stopper/1
9	root	20	0	0	0	0	S	0.0	0.0	0:06.76	ksoftirqd/1
10	root	RT	0	0	0	0	S	0.0	0.0	0:01.77	watchdog/1
11	root	RT	0	0	0	0	S	0.0	0.0	0:00.12	migration/2
12	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	stopper/2
13	root	20	0	0	0	0	S	0.0	0.0	0:01.32	ksoftirqd/2
14	root	RT	0	0	0	0	S	0.0	0.0	0:01.49	watchdog/2
15	root	RT	0	0	0	0	S	0.0	0.0	0:00.14	migration/3
16	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	stopper/3
17	root	20	0	0	0	0	S	0.0	0.0	0:01.35	ksoftirqd/3
18	root	RT	0	0	0	0	S	0.0	0.0	0:01.81	watchdog/3

图6-8 TOP使用截图

统计信息区前五行为系统整体的统计信息。第一行是任务队列信息。第二、三行为进程和 CPU 的信息，当有多个 CPU 时，这些内容可能会超过两行。第四、五行为内存信息。

Top 命令所有的列解释如下所示。

PID: 表示进程 id。

PPID: 表示父进程 id。

UID: 表示进程所有者的用户 id。

USER: 表示进程所有者的用户名。

GROUP: 表示进程所有者的组名。

TTY: 表示启动进程的终端名。不是从终端启动的进程则显示为？。

PR: 表示优先级。

Nice: 负值表示高优先级，正值表示低优先级。

%CPU: 表示上次更新到现在的 CPU 时间占用百分比。

TIME: 表示进程使用的 CPU 时间总计，单位秒。

TIME+: 表示进程使用的 CPU 时间总计，单位 1/100 秒。

%MEM: 表示进程使用的物理内存百分比。

VIRT: 表示进程使用的虚拟内存总量，单位 KB。VIRT=SWAP+RES。

SWAP: 表示进程使用的虚拟内存中，被换出的大小，单位 KB。

RES: 表示进程使用的、未被换出的物理内存大小, 单位 KB。RES=CODE+DATA。

CODE: 表示可执行代码占用的物理内存大小, 单位 KB。

DATA: 表示可执行代码以外的部分(数据段+栈)占用的物理内存大小, 单位 KB。

SHR: 表示共享内存大小, 单位 KB。

nFLT: 表示页面错误次数。

nDRT: 表示最后一次写入到现在, 被修改过的页面数。

HTop

HTop 是 Linux 系统中的一个互动的进程查看器, 一个文本模式的应用程序 (在控制台或者 X 终端中), 需要 ncurses。与 Linux 传统的 top 相比, htop 更加人性化。它可让用户交互式操作, 支持颜色主题, 可横向或纵向滚动浏览进程列表, 并支持鼠标操作。htop 命令还可以显示每个进程的内存实时使用率。它提供了所有进程的常驻内存大小、程序总内存大小、共享库大小等的报告。列表可以水平及垂直滚动。

与 top 相比, htop 有以下优点:

- (1) 可以横向或纵向滚动浏览进程列表, 以便看到所有的进程和完整的命令行;
- (2) 在启动上, 比 top 更快;
- (3) 杀进程时不需要输入进程号;
- (4) htop 支持鼠标操作。

vmstat

vmstat 命令报告关于内核线程的统计信息, 包括处于运行和等待队列、内存、页面调度、磁盘、系统中断、系统调用、上下文切换和 CPU 活动的内核线程。所报告的 CPU 活动是用户方式、系统方式、空闲时间和等待磁盘 I/O 的百分比详细分类。常规的 vmstat 命令输出如清单 6-3 所示。

代码清单 6-3 vmstat 命令输出

```
[root@node1:5 zhoumingyao]# vmstat
procs -----memory----- --swap-- ----io---- --system-- -----cpu-----
 r b  swpd  free    buff  cache   si  so   bi   bo   in  cs us sy id wa st
 1  0  544104 23080516 329112 4670948    0   0    0    1    1    1    0  0 99  0  0
```

如果是虚拟机, Linux 的 vmstat 显示所有虚拟处理器的总 CPU 使用率。无论物理机器、虚拟机, 两者的 vmstat 都有命令行选项可以设定报告的时间间隔(秒级)。如果不指定 vmstat 的报告间隔, 则输出自从系统最近一次启动以来的总 CPU 使用率。如果指定间隔, 统计数据的第一行则是最近一次启动以来所有数据的总和, 不过通常来说都可以忽略不计。如果使用 vmstat 命令时不带任何选项, 或者只带有间隔时间和任意的计数参数, 例如 vmstat 2 10; 那么第一行数字为自系统重新引导以来的平均值。

如果需要检查 CPU 占有率是否很高, Linux 操作系统也可以使用 vmstat 命令查看。我在服务器上运行了一个阶段性占用 CPU 的进程后, 情况如图 6-9 所示。vmstat 输出的第一列是运行队列长度, 值是运行队列中轻量级进程的实际数量。

```
root@facenode4 ~]# vmstat 3 10
```

procs		memory				swap		io		system		cpu			
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa
10	0	0	30956788	345480	454568	0	0	0	0	0	0	1	0	0	100
10	0	0	30956996	345480	454568	0	0	0	0	10426	1113826	34	6	60	0
11	0	0	30957120	345480	454568	0	0	0	32	10480	1087807	28	5	68	0
10	0	0	30957272	345480	454568	0	0	0	0	10377	981240	32	5	63	0
10	0	0	30957404	345484	454568	0	0	0	11	10386	798997	34	4	62	0
10	0	0	30957460	345484	454568	0	0	0	7	10221	829827	35	5	60	0
9	0	0	30957460	345484	454568	0	0	0	0	10291	761259	31	4	65	0
9	0	0	30957460	345484	454568	0	0	0	0	10699	763054	32	4	63	0
11	0	0	30957460	345484	454568	0	0	0	0	10375	770226	34	4	61	0
8	0	0	30957460	345484	454568	0	0	0	0	10568	781352	34	5	61	0

图6-9 检查CPU占有率

具体解释其中每项。

r: 表示运行队列(就是说多少个进程真的分配到 CPU), 当这个值超过了 CPU 数目, 就会出现 CPU 瓶颈了。这个也和 top 的负载有关系, 一般负载超过了 3 就比较高, 超过了 5 就高, 超过了 10 就不正常了, 服务器的状态很危险。top 的负载类似每秒的运行队列。如果运行队列过大, 表示你的 CPU 很繁忙, 一般会造成 CPU 使用率很高。

b: 表示阻塞的进程。

swpd: 表示虚拟内存已使用的大小, 如果大于 0, 表示你的机器物理内存不足了, 如果不是程序内存泄漏的原因, 那么你应该升级内存或者把耗内存的任务迁移到其他机器。

free: 表示空闲的物理内存的大小。

buff: 表示目录里面有什么内容, 权限等的缓存。

cache: 直接用来记忆我们打开的文件, 给文件做缓冲, 把空闲的物理内存的一部分拿来做文件和目录的缓存, 是为了提高程序执行的性能。

si: 每秒从磁盘读入虚拟内存的大小, 如果这个值大于 0, 表示物理内存不够用或者内存泄露了, 要查找消耗内存进程解决掉。

so: 每秒虚拟内存写入磁盘的大小, 如果这个值大于 0, 同上。

bi: 块设备每秒接收的块数量, 这里的块设备是指系统上所有的磁盘和其他块设备, 默认块大小是 1024Byte。

bo: 块设备每秒发送的块数量, 例如我们读取文件, bo 就要大于 0。bi 和 bo 一般都要接近 0, 不然就是 IO 过于频繁, 需要调整。

in: 每秒 CPU 的中断次数, 包括时间中断。

cs: 每秒上下文切换次数, 例如我们调用系统函数, 就要进行上下文切换, 线程的切换, 也要进程上下文切换, 这个值要越小越好, 太大了, 要考虑调低线程或者进程的数目, 例如在 Apache 和 Nginx 这种 Web 服务器中, 我们一般做性能测试时会进行几千并发甚至几万并发的测试, 选择 Web 服务器的进程可以由进程或者线程的峰值一直下调, 压测, 直到 cs 到一个比较小的值, 这个进程和线程数就是比较合适的值了。系统调用也是, 每次调用系统函数, 我们的代码就会进入内核空间, 导致上下文切换, 这个是很耗资源, 也要尽量避免频繁调用系统函数。上下文切换次数过多表示你的 CPU 大部分浪费在上下文切换, 导致 CPU 干正经事的时间少了, CPU 没有充分利用, 是不可取的。

us: 用户 CPU 时间。

sy: 系统 CPU 时间, 如果太高, 表示系统调用时间长, 例如是 I/O 操作频繁。

id: 空闲 CPU 时间, 一般来说, $id+us+sy=100$, 一般我认为 id 是空闲 CPU 使用率, us 是用户 CPU 使用率, sy 是系统 CPU 使用率。

wt: 等待 I/O CPU 时间。

注意, 作为一个 CPU 监视器, vmstat 命令优于 iostat 命令, 因为 vmstat 命令是滚动的, 使得每行的输出更容易扫描, 并且如果有很多磁盘连接到系统中, 由此所涉及的开销更少。下面的示例可以帮助您识别一个程序失控时或 CPU 过度密集以至于不能在一个多用户环境中运行时的情况。

代码清单 6-4 识别程序失控示例

```
# vmstat 2
```

kthr		memory				page				faults				cpu			
r	b	avm	fre	re	pi	po	fr	sr	cy	in	sy	cs	us	sy	id	wa	
1	0	22478	1677	0	0	0	0	0	0	188	1380	157	57	32	0	10	
1	0	22506	1609	0	0	0	0	0	0	214	1476	186	48	37	0	16	
0	0	22498	1582	0	0	0	0	0	0	248	1470	226	55	36	0	9	
2	0	22534	1465	0	0	0	0	0	0	238	903	239	77	23	0	0	
2	0	22534	1445	0	0	0	0	0	0	209	1142	205	72	28	0	0	
2	0	22534	1426	0	0	0	0	0	0	189	1220	212	74	26	0	0	
3	0	22534	1410	0	0	0	0	0	0	255	1704	268	70	30	0	0	
2	1	22557	1365	0	0	0	0	0	0	383	977	216	72	28	0	0	
2	0	22541	1356	0	0	0	0	0	0	237	1418	209	63	33	0	4	
1	0	22524	1350	0	0	0	0	0	0	241	1348	179	52	32	0	16	
1	0	22546	1293	0	0	0	0	0	0	217	1473	180	51	35	0	14	

上面的输出显示了在一个死循环中将程序引入到一个繁忙的多用户系统所带来的效果。头三个报告表明系统平衡在 50%~55%的用户、30%~35%的系统和 10%~15%的 I/O 等待处。当循环程序开始运行, 所有可用的 CPU 周期都被耗用。因为循环程序不进行 I/O, 所以它可以占有前面因为 I/O 等待而未用过的所有周期。更糟的是, 这代表当一个有用进程放弃 CPU 时, 始终有一个进程准备接管 CPU。因为循环程序的优先级与所有其他前台进程一样, 所以当另一个进程变得可分派时它也没必要一定得放弃 CPU。该程序运行大约 10 秒钟 (五个报告), 然后由 Vmstat 命令报告的活动恢复到较正常的模式。

最佳利用是让 CPU 在 100%的时间中工作。这适用于单用户系统的情况, 不需要共享 CPU。总的来说, 如果 us+sy 时间低于 90%, 则不认为单用户系统是 CPU 受限制的。但是, 如果在一个多用户系统中 us+sy 时间超过 80%, 则进程可能要花时间在运行队列中等待。响应时间和吞吐量会受损害。

Sar 工具

sar (System Activity Reporter 系统活动情况报告) 是目前 Linux 上最为全面的系统性能分析工具之一, 可以从多方面对系统的活动进行报告, 包括: 文件的读写情况、系统调用的使用情况、磁盘 I/O、CPU 效率、内存使用状况、进程活动及 IPC 有关的活动等。

命令格式是这样的, sar [options] [-A] [-o file] t [n], 其中各选项代表的意义如下所示。

t 表示采样间隔，n 为采样次数，默认值是 1。

-o file 表示将命令结果以二进制数形式存放在文件中，file 是文件名。

options 为命令行选项。

-A: 所有报告的总和。

-u: 输出 CPU 使用情况的统计信息。

-v: 输出 inode、文件和其他内核表的统计信息。

-d: 输出每一个块设备的活动信息。

-r: 输出内存和交换空间的统计信息。

-b: 显示 I/O 和传送速率的统计信息。

-a: 文件读写情况。

-c: 输出进程统计信息，每秒创建的进程数。

-R: 输出内存页面的统计信息。

-y: 终端设备活动情况。

-w: 输出系统交换活动信息。

例如，每 10 秒采样一次，连续采样 3 次，观察 CPU 的使用情况，并将采样结果以二进制数形式存入当前目录下的文件 test 中，需键入如下命令：

```
sar -u -o test 10 3
```

屏幕显示如清单 6-5 所示。

代码清单 6-5 Sar 命令输出

```
17:06:16 CPU %user %nice %system %iowait %steal %idle
17:06:26 all 0.00 0.00 0.20 0.00 0.00 99.80
17:06:36 all 0.00 0.00 0.20 0.00 0.00 99.80
17:06:46 all 0.00 0.00 0.10 0.00 0.00 99.90
Average: all 0.00 0.00 0.17 0.00 0.00 99.83
```

输出项说明如下。

CPU: all 表示统计信息为所有 CPU 的平均值。

%user: 显示在用户级别(application)运行使用 CPU 总时间的百分比。

%nice: 显示在用户级别，用于 nice 操作，所占用 CPU 总时间的百分比。

%system: 在核心级别(kernel)运行所使用 CPU 总时间的百分比。

%iowait: 显示用于等待 I/O 操作占用 CPU 总时间的百分比。

%steal: 管理程序(hypervisor)为另一个虚拟进程提供服务而等待虚拟 CPU 的百分比。

%idle: 显示 CPU 空闲时间占用 CPU 总时间的百分比。

以上输出结果如果出现以下情况，有对应的症状表示。

- (1) 若%iowait 的值过高，表示硬盘存在 I/O 瓶颈；
- (2) 若%idle 的值高但系统响应慢时，有可能是 CPU 等待分配内存，此时应加大内存容量；
- (3) 若%idle 的值持续低于 1，则系统的 CPU 处理能力相对较低，表明系统中最需要解决的资源是 CPU。

如果要查看二进制文件 test 中的内容，可以通过如下 sar 命令来完成。

```
sar -u -f test
```

mpstat 命令

Linux 还提供命令行工具 mpstat，以列表方式展示每个虚拟处理器的 CPU 使用率。

直接运行 mpstat 命令，输出如下所示。

代码清单 6-6 mpstat 命令输出

```
[root@node1:5 zhoumingyao]# mpstat
Linux 2.6.32-504.el6.x86_64 (node1)      2015 年 09 月 10 日  _x86_64_      (24 CPU)
18 时 34 分 13 秒  CPU      %usr    %nice    %sys %iowait    %irq    %soft    %steal    %guest    %idle
18 时 34 分 13 秒  all       0.48     0.00     0.04    0.01     0.00     0.00     0.00     0.00    99.47
```

清单输出的内容包括了几个字段，我们逐一解释。

%user: 表示处理用户进程所使用 CPU 的百分比。用户进程是用于应用程序（如 Oracle 数据库）的非内核进程。在本示例输出中，用户 CPU 百分比非常低。

%nice: 表示使用 nice 命令对进程进行降级时 CPU 的百分比。在之前的部分中已经对 nice 命令进行了介绍。简单来说，nice 命令更改进程的优先级。

%system: 表示内核进程使用的 CPU 百分比。

%iowait: 表示等待进行 I/O 所使用的 CPU 时间百分比。

%irq: 表示用于处理系统中断的 CPU 百分比。

%soft: 表示用于软件中断的 CPU 百分比。

%idle: 表示 CPU 的空闲时间。

%intr/s: 表示每秒 CPU 接收的中断总数。

用 mpstat 监控每个虚拟处理器的 CPU 使用率，有助于发现应用中是一些线程比其他线程消耗了更多的 CPU 周期，还是应用的所有线程基本平分 CPU 周期。如果是后者，意味着应用的扩展性比较好。

mpstat 命令和 vmstat 之间存在一定的区别，mpstat 可以显示每个处理器的统计，而 vmstat 显示所有处理器的统计。因此，编写糟糕的应用程序（不使用多线程体系结构）可能会运行在一个多处理器机器上，而不使用所有处理器。从而导致一个 CPU 过载，而其他 CPU 却很空闲。通过 mpstat 可以轻松诊断这些类型的问题。

总的来说，mpstat 命令还产生与 CPU 有关的统计信息，因此所有与 CPU 问题有关的讨论也

都适用于 mpstat。当看到较低的%idle 数字时，就知道出现了 CPU 不足的问题。当看到较高的%iowait 数字时，就可以知道在当前负载下 I/O 子系统出现了某些问题。该信息对于解决 Oracle 等关系型数据库的性能问题时非常有效。

pidstat

除 CPU 使用率之外，监控 CPU 调度程序运行队列对于分辨系统是否满负荷也有重要意义。运行队列中就是那些已准备好运行、正等待可用 CPU 的轻量级进程。如果准备运行的轻量级进程数超过系统所能处理的上限，运行队列就会很长。运行队列很长表明系统负载可能已饱和。系统运行队列长度等于虚拟处理器的个数时，用户不会明显感觉到性能下降。此处虚拟处理器的个数就是使用硬件线程的个数，也就是 Java API Runtime.availableProcessors()的返回值。当运行队列长度达到虚拟处理的 4 倍或更多时，系统的响应就非常迟缓了。

一般如果在很长一段时间里，运行队列的长度一直都超过虚拟处理器个数的 1 倍左右需要关注了，只是暂时还不需要立即采取行动。如果在很长一段时间里，运行队列的长度达到虚拟处理器个数的 3~4 倍或更高，则需要立刻引起注意或采取行动。

解决运行队列较长的问题普遍使用两种方法。一种是增加 CPU 以分担负载或减小处理器的负载量，这种方法从根本上减少了每个虚拟处理器上的活动线程数，从而减少了运行队列中的轻量级进程数。另一种方法是分析系统中运行的应用，改进 CPU 使用率。换句话说，研究可以减少应用运行所需 CPU 周期的方法，如减少垃圾收集的频率或采用完成同样任务但 CPU 指令更少的算法。性能专家在减少代码路径长度以及为了改进 CPU 指令选择性时，通常会考虑这种方法，Java 程序员可以通过更有效的算法和数据结构来实现更好的性能。这是因为，虽然现代 JIT 编译器可以产生成熟优化的代码以改善应用性能，但 Java 程序员几乎无法操纵 JIT 编译器，所以应该关注算法和数据结构的效率，通过性能分析可以找出哪些算法和数据结构值得关注。

可以采用 pidstat 工具来检测 CPU 使用，该工具可以具体找出占用 CPU 的线程，如图 6-10 所示。其中使用的-t 参数将系统性能的监控细化到线程级别。

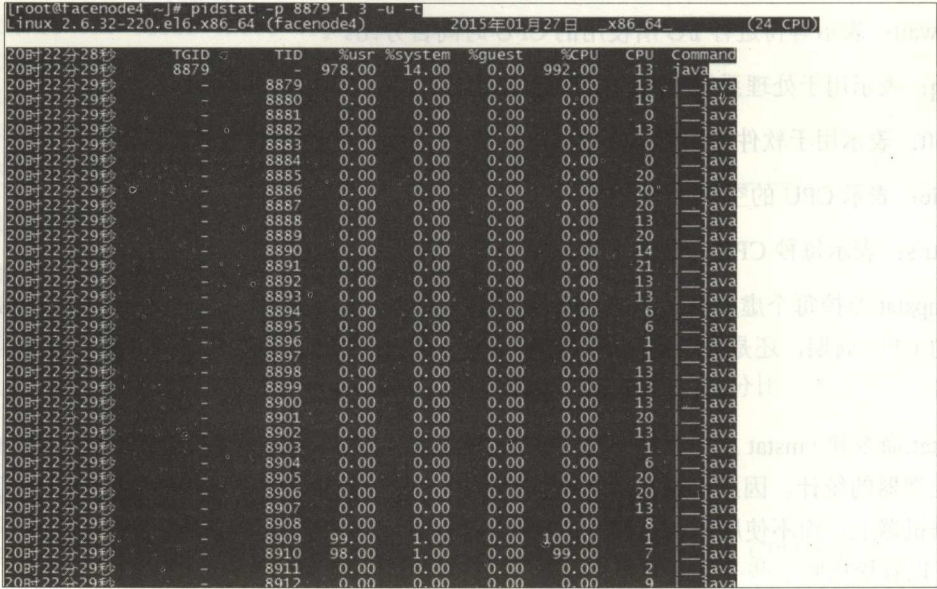


图6-10 pid命令打印

下面这些是输出的解释。

minflt/s: 每秒次缺页错误次数 (minor page faults), 次缺页错误次数意即虚拟内存地址映射成物理内存地址产生的 page fault 次数;

majflt/s: 每秒主缺页错误次数 (major page faults), 当虚拟内存地址映射成物理内存地址时, 相应的 page 在 swap 中, 这样的 page fault 为 major page fault, 一般在内存使用紧张时产生;

VSZ: 该进程使用的虚拟内存 (以 KB 为单位);

RSS: 该进程使用的物理内存 (以 KB 为单位);

%MEM: 该进程使用内存的百分比;

Command: 拉起进程对应的命令。

图 6-11 显示线程 17185 占用了 18.33% 的 CPU 资源。

平均时间:	TGID	TID	%usr	%system	%guest	%CPU	CPU	Command
平均时间:	17157	-	29.00	26.00	0.00	55.00	-	java
平均时间:	-	17157	0.00	0.00	0.00	0.00	-	java
平均时间:	-	17158	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17159	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17160	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17161	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17162	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17163	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17164	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17165	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17166	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17167	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17168	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17169	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17170	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17171	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17172	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17173	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17174	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17175	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17176	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17177	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17178	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17179	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17180	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17181	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17182	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17183	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17184	0.00	0.00	0.00	0.00	-	ava
平均时间:	-	17185	10.00	8.33	0.00	18.33	-	ava
平均时间:	-	17186	1.33	1.00	0.00	2.33	-	ava
平均时间:	-	17187	1.33	1.33	0.00	2.67	-	ava
平均时间:	-	17188	2.33	2.33	0.00	4.67	-	ava
平均时间:	-	17189	8.33	7.33	0.00	15.67	-	ava
平均时间:	-	17190	3.33	2.33	0.00	5.67	-	ava
平均时间:	-	17191	2.33	3.00	0.00	5.33	-	ava
平均时间:	-	17192	0.67	1.00	0.00	1.67	-	ava

图6-11 线程17185

-d 参数表明监控对象为磁盘 I/O。增加了-d 参数后命令输出如下所示。

代码清单 6-7 pidstat -d 运行输出

```
[root@facenode4 ~]# pidstat -p 17685 -d -t 1 3
Linux 2.6.32-220.el6.x86_64 (facenode4)      2015 年 01 月 28 日  _x86_64_      (24 CPU)

11 时 41 分 39 秒      TGID      TID      kB_rd/s      kB_wr/s      kB_ccwr/s      Command
11 时 41 分 40 秒      17685      -          0.00          56.00          0.00      java
11 时 41 分 40 秒      -          17685      0.00          0.00          0.00      |__java
11 时 41 分 40 秒      -          17686      0.00          0.00          0.00      |__java
11 时 41 分 40 秒      -          17687      0.00          0.00          0.00      |__java
11 时 41 分 40 秒      -          17688      0.00          0.00          0.00      |__java
```

11 时 41 分 40 秒	-	17689	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17690	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17691	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17692	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17693	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17694	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17695	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17696	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17697	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17698	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17699	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17700	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17701	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17702	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17703	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17704	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17705	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17706	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17707	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17708	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17709	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17710	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17711	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17712	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17713	0.00	56.00	0.00	__java
11 时 41 分 40 秒	-	17714	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17715	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17716	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17717	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17718	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17719	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17720	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17721	0.00	0.00	0.00	__java
11 时 41 分 40 秒	-	17722	0.00	0.00	0.00	__java

从上面的输出可以看到，线程 17713 每秒产生了大量的磁盘 I/O。

-r 参数可以监控内存使用情况，增加了-r 参数后的输出如清单 6-8 所示。

代码清单 6-8 pidstat 增加-r 参数运行输出

```
[root@facenode4 ~]# pidstat -p 17685 -r -t 1 3
Linux 2.6.32-220.el6.x86_64 (facenode4)      2015 年 01 月 28 日   _x86_64_      (24 CPU)

11 时 44 分 13 秒      TGID      TID minflt/s  majflt/s    VSZ    RSS    %MEM  Command
11 时 44 分 14 秒      17685      -    0.00      0.00 10984544  45616   0.14   java
11 时 44 分 14 秒      -      17685    0.00      0.00 10984544  45616   0.14   |__java
11 时 44 分 14 秒      -      17686    0.00      0.00 10984544  45616   0.14   |__java
11 时 44 分 14 秒      -      17687    0.00      0.00 10984544  45616   0.14   |__java
11 时 44 分 14 秒      -      17688    0.00      0.00 10984544  45616   0.14   |__java
```


11 时 44 分 14 秒	-	17689	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17690	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17691	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17692	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17693	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17694	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17695	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17696	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17697	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17698	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17699	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17700	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17701	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17702	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17703	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17704	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17705	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17706	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17707	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17708	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17709	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17710	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17711	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17712	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17713	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17714	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17715	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17716	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17717	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17718	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17719	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17720	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17721	0.00	0.00	10984544	45616	0.14	__java
11 时 44 分 14 秒	-	17722	0.00	0.00	10984544	45616	0.14	__java

jstack 命令

如果需要查看具体哪个类造成的 CPU 资源消耗较大，可以通过 `jstack -l pid` 命令找到它们。

`jstack` 命令执行的输出结果如清单 6-9 所示。

代码清单 6-9 jstack 命令执行结果

```
[root@facenode4 zhoumingyao]# cat log.txt
2015-01-28 11:15:13
Full thread dump Java HotSpot(TM) 64-Bit Server VM (20.45-b01 mixed mode):

"Attach Listener" daemon prio=10 tid=0x00007fde20001000 nid=0x438d waiting on
condition [0x0000000000000000]
    java.lang.Thread.State: RUNNABLE
```

```
Locked ownable synchronizers:
  - None

"DestroyJavaVM" prio=10 tid=0x00007fde7c006800 nid=0x4306 waiting on condition
[0x0000000000000000]
  java.lang.Thread.State: RUNNABLE

Locked ownable synchronizers:
  - None

"Thread-9" prio=10 tid=0x00007fde7c119800 nid=0x432a waiting for monitor entry
[0x00007fde59037000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at java.io.PrintStream.println(PrintStream.java:727)
    - waiting to lock <0x000000060b002148> (a java.io.PrintStream)
    at HoldCPUMain$HoldCPUTask.run(HoldCPUMain.java:10)
    at java.lang.Thread.run(Thread.java:662)

Locked ownable synchronizers:
  - None

"Thread-8" prio=10 tid=0x00007fde7c117800 nid=0x4329 waiting for monitor entry
[0x00007fde59138000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at java.io.PrintStream.println(PrintStream.java:727)
    - waiting to lock <0x000000060b002148> (a java.io.PrintStream)
    at HoldCPUMain$HoldCPUTask.run(HoldCPUMain.java:10)
    at java.lang.Thread.run(Thread.java:662)

Locked ownable synchronizers:
  - None

"Thread-7" prio=10 tid=0x00007fde7c115800 nid=0x4328 waiting for monitor entry
[0x00007fde59239000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at java.io.PrintStream.println(PrintStream.java:727)
    - waiting to lock <0x000000060b002148> (a java.io.PrintStream)
    at HoldCPUMain$HoldCPUTask.run(HoldCPUMain.java:10)
    at java.lang.Thread.run(Thread.java:662)

Locked ownable synchronizers:
  - None

"Thread-6" prio=10 tid=0x00007fde7c113800 nid=0x4327 waiting for monitor entry
[0x00007fde5933a000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at java.io.PrintStream.println(PrintStream.java:727)
```



```
- waiting to lock <0x000000060b002148> (a java.io.PrintStream)
at HoldCPUMain$HoldCPUTask.run(HoldCPUMain.java:10)
at java.lang.Thread.run(Thread.java:662)
```

Locked ownable synchronizers:

- None

```
"Thread-5" prio=10 tid=0x00007fde7c111800 nid=0x4326 waiting for monitor entry
[0x00007fde5943b000]
```

java.lang.Thread.State: BLOCKED (on object monitor)

```
at java.io.PrintStream.println(PrintStream.java:727)
- waiting to lock <0x000000060b002148> (a java.io.PrintStream)
at HoldCPUMain$HoldCPUTask.run(HoldCPUMain.java:10)
at java.lang.Thread.run(Thread.java:662)
```

Locked ownable synchronizers:

- None

```
"Thread-4" prio=10 tid=0x00007fde7c10f800 nid=0x4325 waiting for monitor entry
[0x00007fde5953c000]
```

java.lang.Thread.State: BLOCKED (on object monitor)

```
at java.io.PrintStream.println(PrintStream.java:727)
- waiting to lock <0x000000060b002148> (a java.io.PrintStream)
at HoldCPUMain$HoldCPUTask.run(HoldCPUMain.java:10)
at java.lang.Thread.run(Thread.java:662)
```

Locked ownable synchronizers:

- None

```
"Thread-3" prio=10 tid=0x00007fde7c10d800 nid=0x4324 waiting for monitor entry
[0x00007fde5963d000]
```

java.lang.Thread.State: BLOCKED (on object monitor)

```
at java.io.PrintStream.println(PrintStream.java:727)
- waiting to lock <0x000000060b002148> (a java.io.PrintStream)
at HoldCPUMain$HoldCPUTask.run(HoldCPUMain.java:10)
at java.lang.Thread.run(Thread.java:662)
```

Locked ownable synchronizers:

- None

```
"Thread-2" prio=10 tid=0x00007fde7c10b800 nid=0x4323 runnable [0x00007fde5973e000]
```

java.lang.Thread.State: RUNNABLE

```
at java.io.FileOutputStream.writeBytes(Native Method)
at java.io.FileOutputStream.write(FileOutputStream.java:282)
at
```

```
java.io.BufferedOutputStream.flushBuffer(BufferedOutputStream.java:65)
```

```
at java.io.BufferedOutputStream.flush(BufferedOutputStream.java:123)
```

```
- locked <0x000000060b02c530> (a java.io.BufferedOutputStream)
```

```

at java.io.PrintStream.write(PrintStream.java:432)
- locked <0x000000060b002148> (a java.io.PrintStream)
at sun.nio.cs.StreamEncoder.writeBytes(StreamEncoder.java:202)
at sun.nio.cs.StreamEncoder.implFlushBuffer(StreamEncoder.java:272)
at sun.nio.cs.StreamEncoder.flushBuffer(StreamEncoder.java:85)
- locked <0x000000060b00cfa0> (a java.io.OutputStreamWriter)
at java.io.OutputStreamWriter.flushBuffer(OutputStreamWriter.java:168)
at java.io.PrintStream.newLine(PrintStream.java:496)
- locked <0x000000060b002148> (a java.io.PrintStream)
at java.io.PrintStream.println(PrintStream.java:729)
- locked <0x000000060b002148> (a java.io.PrintStream)
at HoldCPUMain$HoldCPUTask.run(HoldCPUMain.java:10)
at java.lang.Thread.run(Thread.java:662)

```

Locked ownable synchronizers:

- None

"Thread-1" prio=10 tid=0x00007fde7c109800 nid=0x4322 waiting for monitor entry [0x00007fde5983f000]

```

java.lang.Thread.State: BLOCKED (on object monitor)
at java.io.PrintStream.println(PrintStream.java:727)
- waiting to lock <0x000000060b002148> (a java.io.PrintStream)
at HoldCPUMain$HoldCPUTask.run(HoldCPUMain.java:10)
at java.lang.Thread.run(Thread.java:662)

```

Locked ownable synchronizers:

- None

"Thread-0" prio=10 tid=0x00007fde7c108000 nid=0x4321 waiting for monitor entry [0x00007fde59940000]

```

java.lang.Thread.State: BLOCKED (on object monitor)
at java.io.PrintStream.println(PrintStream.java:727)
- waiting to lock <0x000000060b002148> (a java.io.PrintStream)
at HoldCPUMain$HoldCPUTask.run(HoldCPUMain.java:10)
at java.lang.Thread.run(Thread.java:662)

```

Locked ownable synchronizers:

- None

"Low Memory Detector" daemon prio=10 tid=0x00007fde7c0d2800 nid=0x431f runnable [0x0000000000000000]

```

java.lang.Thread.State: RUNNABLE

```

Locked ownable synchronizers:

- None

"C2 CompilerThread1" daemon prio=10 tid=0x00007fde7c0d0000 nid=0x431e waiting on condition [0x0000000000000000]


```

java.lang.Thread.State: RUNNABLE

Locked ownable synchronizers:
- None

"C2 CompilerThread0" daemon prio=10 tid=0x00007fde7c0cd000 nid=0x431d waiting on
condition [0x0000000000000000]
  java.lang.Thread.State: RUNNABLE

  Locked ownable synchronizers:
    - None

"Signal Dispatcher" daemon prio=10 tid=0x00007fde7c0cb000 nid=0x431c runnable
[0x0000000000000000]
  java.lang.Thread.State: RUNNABLE

  Locked ownable synchronizers:
    - None

"Finalizer" daemon prio=10 tid=0x00007fde7c0af000 nid=0x431b in Object.wait()
[0x00007fde59f46000]
  java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    - waiting on <0x000000060b002098> (a java.lang.ref.ReferenceQueue$Lock)
    at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:118)
    - locked <0x000000060b002098> (a java.lang.ref.ReferenceQueue$Lock)
    at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:134)
    at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:171)

  Locked ownable synchronizers:
    - None

"Reference Handler" daemon prio=10 tid=0x00007fde7c0ad000 nid=0x431a in
Object.wait() [0x00007fde5a047000]
  java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    - waiting on <0x000000060b002138> (a java.lang.ref.Reference$Lock)
    at java.lang.Object.wait(Object.java:485)
    at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:116)
    - locked <0x000000060b002138> (a java.lang.ref.Reference$Lock)

  Locked ownable synchronizers:
    - None

"VM Thread" prio=10 tid=0x00007fde7c0a6000 nid=0x4319 runnable

"GC task thread#0 (ParallelGC)" prio=10 tid=0x00007fde7c019800 nid=0x4307 runnable

```

```
"GC task thread#1 (ParallelGC)" prio=10 tid=0x00007fde7c01b000 nid=0x4308 runnable
"GC task thread#2 (ParallelGC)" prio=10 tid=0x00007fde7c01d000 nid=0x4309 runnable
"GC task thread#3 (ParallelGC)" prio=10 tid=0x00007fde7c01f000 nid=0x430a runnable
"GC task thread#4 (ParallelGC)" prio=10 tid=0x00007fde7c020800 nid=0x430b runnable
"GC task thread#5 (ParallelGC)" prio=10 tid=0x00007fde7c022800 nid=0x430c runnable
"GC task thread#6 (ParallelGC)" prio=10 tid=0x00007fde7c024800 nid=0x430d runnable
"GC task thread#7 (ParallelGC)" prio=10 tid=0x00007fde7c026000 nid=0x430e runnable
"GC task thread#8 (ParallelGC)" prio=10 tid=0x00007fde7c028000 nid=0x430f runnable
"GC task thread#9 (ParallelGC)" prio=10 tid=0x00007fde7c02a000 nid=0x4310 runnable
"GC task thread#10 (ParallelGC)" prio=10 tid=0x00007fde7c02b800 nid=0x4311 runnable
"GC task thread#11 (ParallelGC)" prio=10 tid=0x00007fde7c02d800 nid=0x4312 runnable
"GC task thread#12 (ParallelGC)" prio=10 tid=0x00007fde7c02f800 nid=0x4313 runnable
"GC task thread#13 (ParallelGC)" prio=10 tid=0x00007fde7c031000 nid=0x4314 runnable
"GC task thread#14 (ParallelGC)" prio=10 tid=0x00007fde7c033000 nid=0x4315 runnable
"GC task thread#15 (ParallelGC)" prio=10 tid=0x00007fde7c035000 nid=0x4316 runnable
"GC task thread#16 (ParallelGC)" prio=10 tid=0x00007fde7c036800 nid=0x4317 runnable
"GC task thread#17 (ParallelGC)" prio=10 tid=0x00007fde7c038800 nid=0x4318 runnable
"VM Periodic Task Thread" prio=10 tid=0x00007fde7c0e5000 nid=0x4320 .waiting on
condition
JNI global references: 969
```

从上面输出可以看到，观察的应用程序是一个多线程程序，其中有一个线程（Thread-2）正在运行程序。

综合上面所有描述的工作使用及输出情况，我们知道实际程序运行过程中，我们可以总结出来，一般来说，如果 CPU 时钟周期被用于执行操作系统或内核代码，这部分时钟周期就无法用于执行应用程序。因此，改善应用程序性能的策略之一是减少消耗在系统或内核 CPU 上的时钟周期数。但是，这一策略不适用于在系统或内核态上消耗时间极少的应用程序。监控操作系统在系统或内核态上 CPU 的使用情况能够为决策是否采用该策略提供依据。此外，待运行线程在处理器之间的迁移也会导致性能下降。大多数操作系统的 CPU 调度程序会将待运行线程分配给上次运行它

的虚拟处理器。如果这个虚拟处理器忙，调度程序就会将待运行线程迁移到其他可用的虚拟处理器。线程迁移会对应用性能造成影响，这是因为新的虚拟处理器缓存中可能没有待运行线程所需的数据或状态信息。多核系统上运行 Java 应用可能会产生大量的线程迁移，减少迁移的策略是创建处理器组并将 Java 应用分配给这些处理器组。一般性准则是，如果横跨多核或虚拟处理器的 Java 应用每秒迁移超过 500 次，将 Java 应用绑定在处理器上有益处。

6.1.2 监控内存

除了 CPU 使用率，还需要监控系统内存相关的属性，例如页面调度或页面交换、加锁、线程迁移中的让步式和抢占式上下文切换。

系统在进行页面交换或使用虚拟内存时，Java 应用或 JVM 会表现出明显的性能问题。当应用运行所需的内存超过可用物理内存时，就会发生页面交换。为了应对这种可能出现的情况，通常要为系统配置 swap 空间。swap 空间一般会在一个独立的磁盘分区上。当应用耗尽物理内存时，操作系统会将应用的一部分置换到磁盘上的 swap 空间，通常是应用中最少运行的部分，以免影响整个应用或者应用最忙的那部分。当访问应用中被置换出去的部分时，就必须将它从磁盘置换进内存，而这种置换活动会对应用的响应性和吞吐量造成很大影响。

此外，JVM 垃圾收集器在系统页面交换时的性能也很差，这是由于垃圾收集器为了回收不可达对象所占用的空间，需要访问大量的内存。如果 Java 堆的一部分被置换出去，就必须先置换进内存以便垃圾收集器扫描存活对象，这会增加垃圾收集的持续时间。垃圾收集时一种 Stop-The-World（时空停止）操作，即停止所有正在运行的应用线程，如果此时系统正在进行页面交换，则会引起 JVM 长时间的停顿。如果出现垃圾收集时间变长，系统有可能正在进行页面交换。为了验证这一点，你必须监控系统的页面交换。

我们本小节的监控方式演示都会按照 Windows 篇和 Linux 篇，其中 Windows 针对的是 Windows7 操作系统，Linux 针对的是 CentosV6.5 操作系统。

6.1.2.1 内存使用率监控工具——Windows 篇

前面在介绍 CPU 监控的时候，提到过 Windows 操作系统上面有一个工具 Perfmon，同样该工具也可以被用来监控内存使用情况。如表 6-3 所示，罗列了和内存监控相关的计数器，读者可以自由添加，具体使用方式详见 6.1.1CPU 使用率监控工具之 Windows 篇。

表 6-3 Perfmon 内存监控功能

性能对象	计数器	提供的信息
Memory	Available Bytes	Available Bytes 显示出当前空闲的物理内存总量。当这个数值变小时，Windows 开始频繁地调用磁盘页面文件。如果这个数值很小，例如小于 5 MB，系统会将大部分时间消耗在操作页面文件上
Memory	% Committed Bytes in Use	% Committed Bytes In Use 是 Memory: Committed Bytes 与 Memory: Commit Limit 之间的比值。（Committed memory 指如果需要写入磁盘时已在分页文件中保留空间的处于使用中的物理内存。Commit Limit 是由分页文件的大小而决定的。如果扩大了分页文件，该比例就会减小）。这个计数器只显示当前百分比；而不是一个平均值

续表

性能对象	计数器	提供的信息
Memory	Page Faults/sec	Page Faults/sec 是指处理器处理错误页的综合速率。用错误页数/秒来计算。当处理器请求一个不在其工作集（在物理内存中的空间）内的代码或数据时出现的页错误。这个计数器包括硬错误（那些需要磁盘访问的）和软错误（在物理内存的其他地方找到的错误页）。许多处理器可以在有大量软错误的情况下继续操作。但是，硬错误可以导致明显的拖延。这个计数器显示用上两个实例中观察到的值之间的差除以实例间隔的持续时间所得的值

Perfmon 工具监控每秒内存页面调度(\Memory\Pages/Second)、可用内存字节数 (Memory\Available MBytes)，可以判别系统是否正在进行页面交换。当可用内存变少，并且有页面调度时，系统可能正在进行页面交换。

显示 Windows 页面交换的最简单方式是 typeperf 命令。下面的 typeperf 表示每间隔 5 秒输出可用内存和页面调度(-si 指定报告间隔)。

```
typeperf -si 5 "\Memory\Available Mbytes" "\Memory\Pages/sec"
```

6.1.2.2 内存使用率监控工具——Linux 篇

在做 Linux 系统优化的时候，物理内存是其中最重要的一方面。自然 Linux 也提供了非常多的方法来监控宝贵的内存资源使用情况。下面各项命令逐一列出了 Linux 系统下通过视图工具或命令行来查看内存使用情况的各种方法。

free 命令

free 命令是一个快速查看内存使用情况的方法，它是对/proc/meminfo 收集到的信息的一个概述。运行 free 命令，输出内容如下所示。

代码清单 6-10 free 命令

```
[root@nodel:1 zhoumingyao]# free
              total        used        free      shared    buffers     cached
Mem:      32830232    23849720    8980512         6816     346804    17367556
-/+ buffers/cache:    6135360    26694872
Swap:      8191996     543048     7648948
```

各输出列的解释如下所示。

- total: 总计物理内存的大小;
- used: 已使用多大;
- free: 可用有多少;
- Shared: 多个进程共享的内存总额;
- Buffers/cached: 磁盘缓存的大小;
- 第三行(-/+ buffers/cache):
- used: 已使用多大;

free: 可用有多少。

第二行(mem)的 used/free 与第三行(-/+ buffers/cache) used/free 存在区别。这两个输出的区别在于从使用的角度来看, 第一行是从操作系统的角度来看, 因为对于操作系统而言, buffers/cached 都是属于被使用的, 所以可用内存是 16176KB, 已用内存是 3250004KB, 其中包括, 内核(操作系统)使用+Application(X, oracle,etc)使用的+buffers+cached。第三行所指的是从应用程序角度来看, 对于应用程序来说, buffers/cached 是等于可用的内存, 因为 buffer/cached 是为了提高文件读取的性能, 当应用程序需在用到内存的时候, buffer/cached 会很快地被回收。所以从应用程序的角度来说, 可用内存=系统 free memory+buffers+cached。

/proc/meminfo

Linux 系统上的/proc 目录是一种文件系统, 即 proc 文件系统。与其他常见的文件系统不同的是, /proc 是一种伪文件系统(即虚拟文件系统), 存储的是当前内核运行状态的一系列特殊文件, 用户可以通过这些文件查看有关系统硬件及当前正在运行进程的信息, 甚至可以通过更改其中某些文件来改变内核的运行状态。基于/proc 文件系统如上所述的特殊性, 其内的文件也常被称作虚拟文件, 并具有一些独特的特点。例如, 其中有些文件虽然使用查看命令查看时会返回大量信息, 但文件本身的大小却会显示为 0 字节。此外, 这些特殊文件中大多数文件的时间及日期属性通常为当前系统时间和日期, 这跟它们随时会被刷新(存储于 RAM 中)有关。为了查看及使用上的方便, 这些文件通常会按照相关性进行分类存储于不同的目录甚至子目录中, 如/proc/scsi 目录中存储的就是当前系统上所有 SCSI 设备的相关信息, /proc/N 中存储的则是系统当前正在运行的进程的相关信息, 其中 N 为正在运行的进程(可以想象得到, 在某进程结束后其相关目录则会消失)。

查看内存使用情况最简单的方法是通过/proc/meminfo, 这个动态更新的虚拟文件实际上是许多其他内存相关工具(如: free / ps / top)等的组合显示。/proc/meminfo 列出了所有你想了解的内存的使用情况。

代码清单 6-11 查看 meminfo

```
[root@node1:1 zhoumingyao]# cat /proc/meminfo
MemTotal:      32830232 kB
MemFree:       7727192 kB
Buffers:       346620 kB
Cached:       17367504 kB
SwapCached:    153312 kB
Active:        8054688 kB
Inactive:     16041548 kB
Active(anon):  5834084 kB
Inactive(anon): 554848 kB
Active(file):  2220604 kB
Inactive(file): 15486700 kB
Unevictable:    0 kB
Mlocked:       0 kB
SwapTotal:     8191996 kB
SwapFree:      7648948 kB
Dirty:         160 kB
Writeback:     0 kB
AnonPages:     6271868 kB
```

Mapped:	115688 kB
Shmem:	6816 kB
Slab:	671440 kB
SReclaimable:	604084 kB
SUnreclaim:	67356 kB
KernelStack:	10936 kB
PageTables:	77952 kB
NFS_Unstable:	0 kB
Bounce:	0 kB
WritebackTmp:	0 kB
CommitLimit:	24607112 kB
Committed_AS:	12565524 kB
VmallocTotal:	34359738367 kB
VmallocUsed:	337008 kB
VmallocChunk:	34342462520 kB
HardwareCorrupted:	0 kB
AnonHugePages:	5605376 kB
HugePages_Total:	0
HugePages_Free:	0
HugePages_Rsvd:	0
HugePages_Surp:	0
Hugepagesize:	2048 kB
DirectMap4k:	5056 kB
DirectMap2M:	2045952 kB
DirectMap1G:	31457280 kB

逐一解释清单中的字段所代表的意义。

- MemTotal: 所有可用 RAM 大小（即物理内存减去一些预留位和内核的二进制代码大小）;
- MemFree: LowFree 与 HighFree 的总和，被系统留着未使用的内存;
- Buffers: 用来给文件做缓冲大小;
- Cached: 被高速缓冲存储器（cache memory）用的内存的大小（等于 diskcache minus SwapCache）;
- SwapCached: 被高速缓冲存储器（cache memory）用的交换空间的大小，已经被交换出来的内存，但仍然被存放在 swapfile 中。用来在需要的时候很快被替换而不需要再次打开 I/O 端口。
- Active: 在活跃使用中的缓冲或高速缓冲存储器页面文件的大小，除非非常必要否则不会被移作他用;
- Inactive: 在不经常使用中的缓冲或高速缓冲存储器页面文件的大小，可能被用于其他途径;
- HighFree: 该区域不是直接映射到内核空间。内核必须使用不同的手法使用该段内存;
- LowFree: 低位可以达到高位内存一样的作用，而且它还能够被内核用来记录一些自己的数据结构。
- SwapTota: 交换空间的总大小;
- SwapFree: 未被使用交换空间的大小;

- Dirty: 等待被写回到磁盘的内存大小;
- Writeback: 正在被写回到磁盘的内存大小;
- AnonPages: 未映射页的内存大小;
- Mapped: 设备和文件等映射的大小;
- Slab: 内核数据结构缓存的大小, 可以减少申请和释放内存带来的消耗;
- SReclaimable: 可收回 Slab 的大小;

SUnreclaim: 不可收回 Slab 的大小 (SUnreclaim+SReclaimable=Slab);

PageTables: 管理内存分页页面的索引表的大小;

NFS_Unstable:不稳定页表的大小。

JMap 方式

JMap 是一个可以输出所有内存中对象的工具, 甚至可以将 VM 中的 heap, 以二进制数输出成文本。使用命令 SHELL `jmap -histo pid > a.log` 可以将输出重定向并保存到文本文件中去 (Windows 下也可以使用), 相应地, 可以使用文本对比工具可以打开盖文件, 并查看 GC 回收了哪些对象。

代码清单 6-12 JMap 命令输出

```
[root@node1:10 zhoumingyao]# jmap -histo 19940
num      #instances      #bytes  class name
-----
 1:         1177        74434696  [I
 2:        10357        22517920  [B
 3:         2875         240312   [C
 4:         1828        124976   [Ljava.lang.Object;
 5:          519         59288   java.lang.Class
 6:         2324         55776   java.lang.String
 7:         1344         53760   java.lang.ref.Finalizer
 8:         1377         44064   java.io.File
 9:         1327         42464   java.io.FileDescriptor
10:         1521         24336   java.lang.Object
11:          664         21248   java.io.FileInputStream
12:          663         21216   java.io.FileOutputStream
13:          640         20480   java.util.Vector
14:           45         16920   java.lang.Thread
15:          661        10576   java.io.FileOutputStream$1
16:          125          9000   java.lang.reflect.Field
17:          102          8624   [Ljava.lang.String;
18:          257          8224   java.util.HashMap$Node
19:           20          5312   [Ljava.util.HashMap$Node;
20:          128          5120   java.lang.ref.SoftReference
21:          258          4128   java.lang.Integer
22:          114          3648   java.util.Hashtable$Entry
23:          151          3624   java.lang.StringBuilder
24:           81          2592   java.util.concurrent.ConcurrentHashMap$Node
25:           5           2408   [J
```

26:	29	2320	[Ljava.lang.ThreadLocal\$ThreadLocalMap\$Entry;
27:	18	2144	[Ljava.util.Hashtable\$Entry;
28:	49	1960	java.security.AccessControlContext
29:	33	1848	sun.nio.cs.UTF_8\$Encoder
30:	38	1824	sun.util.locale.LocaleObjectCache\$CacheEntry
31:	1	1544	[[B
32:	23	1472	java.net.URL
33:	25	1200	java.util.HashMap
34:	14	1120	java.lang.reflect.Constructor

一个 heap dump 是 Java 虚拟机 (JVM) 在某一时刻所有对象的快照。JVM 从堆中为所有的类实例和数组分配内存。当一个对象不再被使用并且没有对它的引用时, 垃圾回收器回收其堆内存。通过查看堆, 你可以找到对象创建的位置, 发现对象的引用。

下面的命令可以将 19940 进程的内存 heap 输出出来到 f1 文件里。

代码清单 6-13 jmap 内存堆输出到文件

```
[root@node1:10 zhoumingyao]# jmap -dump:format=b,file=f1 19940
Dumping heap to /home/zhoumingyao/f1 ...
Heap dump file created
```

生成的 f1 文件是一个二进制文件, 需要用特定的工具才能打开。例如 JHat 工具, 如下所示。

代码清单 6-14 JHat 命令输出

```
[root@node1:1 zhoumingyao]# jhat f1
Reading from f1...
Dump file created Tue Sep 15 11:09:56 CST 2015
Snapshot read, resolving...
Resolving 29677 objects...
Chasing references, expect 5 dots.....
Eliminating duplicate references.....
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

以上这些输出显示已经打开了一个 HTTP 服务, 端口号为 7000, 我们打开浏览器, 输入地址: http://ip:7000, 然后点击访问, 页面如图 6-12 所示。

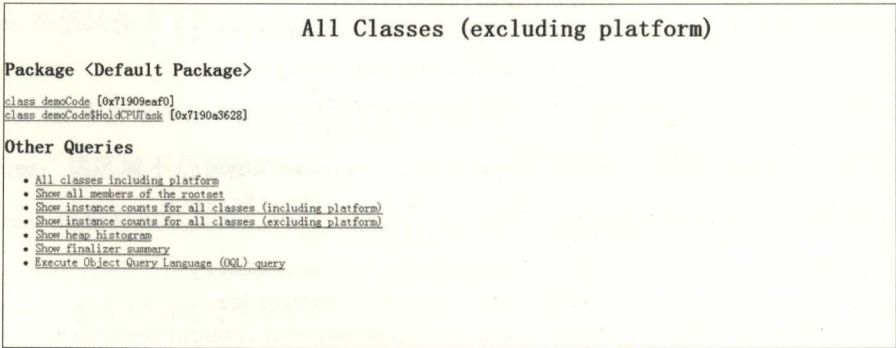


图6-12 JHat UI1

我们可以查看各类信息，比如点击 “All Classes including platform”，可以查看整个应用程序里面包含的类清单，如图 6-13 所示。

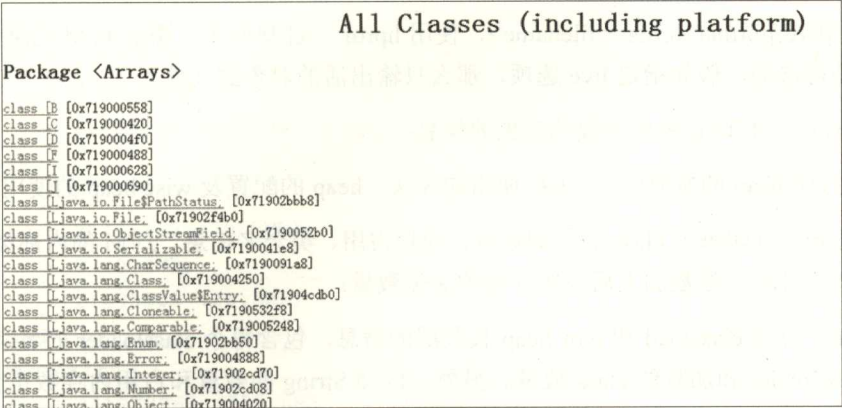


图6-13 JHat UI2

JDK 自带的 JHat 命令可以被用来分析 java 堆的命令，可以将堆中的对象以 html 的形式显示出来，包括对象的数量、大小等，并支持对象查询语言。如果 dump 出来的堆很大，在启动时会报堆空间不足的错误，可以使用如下参数，jhat -J-Xmx512m <heap dump file>。

除了 JHat 之外，我们也可以使用 VisualVM 浏览 heap dump 文件的内容，从而快速查看在堆中分配的对象。Heap dumps 在主窗口的 heap dump 子标签页中显示。你可以打开保存在本地的 heap dump 文件（.hprof）或者使用 VisualVM 捕获正在运行的程序的 heap dumps。如果 JVM 试图从堆中移除不再需要的对象时失败了，VisualVM 可以定位到离该对象最近的垃圾回收根（garbage collecting root）。

第 3 章中曾经讲到虚引用的 finalization 过程，JMap 命令也可以显示有多少对象正在 finalization 队列中，等待 finalizer thread 进行 finalizer。以下命令显示进程 19940 下面有多少个对象在收集队列中。

代码清单 6-15 JMap 输出队列信息

```
[root@node1:1 zhoumingyao]# jmap -finalizerinfo 19940
Attaching to process ID 19940, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.45-b02
Number of objects pending for finalization: 0
```

JMap 命令参数说明：

1) options:

executable Java executable from which the core dump was produced.

（可能是产生 core dump 的 java 可执行程序）

core: 将被打印信息的 core dump 文件；

remote-hostname-or-IP: 远程 debug 服务的主机名或 ip；

server-id: 唯一 id, 假如一台主机上多个远程 debug 服务。

2) 基本参数:

-dump: [live,]format=b,file=<filename>, 使用 hprof 二进制形式, 输出 JVM 的堆内容到文件, live 子选项是可选的, 假如指定 live 选项, 那么只输出活的对象到文件;

-finalizerinfo: 打印正等候回收的对象的信息;

-heap: 打印 heap 的概要信息、GC 使用的算法、heap 的配置及 wise heap 的使用情况;

-histo[:live]: 打印每个 class 的实例数目、内存占用、类全名信息、JVM 的内部类名字开头会加上前缀“*”, live 子参数加上后只统计活的对象数量;

-permstat: 打印 classload 和 jvm heap 长久层的信息, 包含每个 classloader 的名字、活泼性、地址、父 classloader 和加载的 class 数量。另外, 内部 String 的数量和占用内存数也会打印出来;

-F: 强迫在 pid 没有相应的时候使用 -dump 或者 -histo 参数。在这个模式下 live 子参数无效;

-h|-help: 打印辅助信息;

-J: 传递参数给 jmap 启动的 jvm。

如果 pid 需要被打印相关的信息, 最常用的 JMap 使用方式是打印堆内存里面的具体使用情况, 输出如清单 6-16 所示。

代码清单 6-16 JMap 输出堆内存信息

```
[root@node1:1 zhoumingyao]# jmap -heap 19940
Attaching to process ID 19940, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.45-b02
using thread-local object allocation.
Parallel GC with 18 thread(s)
Heap Configuration:
  MinHeapFreeRatio      = 0
  MaxHeapFreeRatio      = 100
  MaxHeapSize           = 8405385216 (8016.0MB)
  NewSize                = 175112192 (167.0MB)
  MaxNewSize            = 2801795072 (2672.0MB)
  OldSize                = 351272960 (335.0MB)
  NewRatio               = 2
  SurvivorRatio          = 8
  MetaspaceSize          = 21807104 (20.796875MB)
  CompressedClassSpaceSize = 1073741824 (1024.0MB)
  MaxMetaspaceSize       = 17592186044415 MB
  G1HeapRegionSize       = 0 (0.0MB)
Heap Usage:
PS Young Generation
Eden Space:
  capacity = 132120576 (126.0MB)
```



```
used      = 97788272 (93.25816345214844MB)
free      = 34332304 (32.74183654785156MB)
74.01441543821305% used

From Space:
  capacity = 21495808 (20.5MB)
  used     = 0 (0.0MB)
  free     = 21495808 (20.5MB)
  0.0% used

To Space:
  capacity = 21495808 (20.5MB)
  used     = 0 (0.0MB)
  free     = 21495808 (20.5MB)
  0.0% used

PS Old Generation
  capacity = 351272960 (335.0MB)
  used     = 0 (0.0MB)
  free     = 351272960 (335.0MB)
  0.0% used

854 interned Strings occupying 55992 bytes.
```

以上输出打印了 JVM 对于堆内存的配置情况、新生代及老生代的配置情况等，具体概念性知识会在第 7 章详细解释。

atop

这是一款用于监控 Linux 系统资源与进程的工具，它以一定的频率记录系统的运行状态，所采集的数据包含系统资源（CPU、内存、磁盘和网络）使用情况和进程运行情况，并能以日志文件的方式保存在磁盘中，服务器出现问题后，我们可获取相应的 atop 日志文件进行分析，如图 6-14 所示。atop 是一款开源软件，我们可以从 <http://www.atoptool.nl/downloadatop.php> 获得其源码和 rpm 安装包。

ATOP - LX										2011/12/17 21:35:38		-----		10m0s elapsed	
PRC	sys	5m13s	user	5m02s	#proc	165	#zombie	1	#exit	269					
CPU	sys	51%	user	50%	irq	1%	idle	98%	wait	1%					
cpu	sys	25%	user	24%	irq	1%	idle	49%	cpu000 w	1%					
cpu	sys	25%	user	25%	irq	0%	idle	49%	cpu001 w	0%					
CPL	avg1	0.14	avg5	0.14	avg15	0.17	csw	2295130	intr	654283					
MEM	tot	1.9G	free	246.9M	cache	717.1M	buff	169.8M	slab	80.0M					
WAP	tot	978.0M	free	975.9M			vmstat	3.5G	vmstat	1.9G					
PAG	scan	38953	stall	0			swin	0	swout	16					
DSK		sda	busy	3%	read	369	write	2607	avio	5.34 ms					
NET	transport		tcp1	1016	tcpo	1238	udpi	165	udpo	165					
NET	network		ipi	1181	ipo	1439	ipfrw	0	deliv	1181					
NET	wlan0	----	pck1	1184	pcko	1442	si	6 Kbps	so	4 Kbps					
PID	SYSCPU	USRCPU	VGROW	RGROW	RDDSK	WRDSK	ST	EXC	S	CPU	CMD	1/49			
4600	2m25s	1m41s	-64K	-1964K	19660K	24K	--	--	R	41%	chrome				
1143	2m11s	88.20s	8712K	8644K	0K	0K	--	--	S	36%	Xorg				
1729	9.31s	25.15s	1412K	164K	0K	16K	--	--	S	6%	compiz				
5523	2.35s	21.72s	0K	0K	-	-	NE	0	E	4%	<firefox>				
4485	6.68s	16.90s	32K	524K	20K	10436K	--	--	S	4%	chrome				
4607	4.26s	19.20s	15572K	4716K	0K	0K	--	--	S	4%	chrome				
1736	6.78s	7.73s	0K	28K	0K	0K	--	--	S	2%	pulseaudio				
4804	0.79s	8.73s	0K	316K	0K	0K	--	--	S	2%	chrome				
2647	0.33s	1.69s	144K	392K	0K	212K	--	--	S	0%	gnome-terminal				

图6-14 atop输出

ATOP 列：该列显示了主机名、信息采样日期和时间点；

PRC 列：该列显示进程整体运行情况，其中

(1) sys、usr 字段分别指示进程在内核态和用户态的运行时间

(2) #proc 字段指示进程总数

(3) #zombie 字段指示僵死进程的数量

(4) #exit 字段指示 atop 采样周期期间退出的进程数量

CPU 列：该列显示 CPU 整体（即多核 CPU 作为一个整体 CPU 资源）的使用情况，我们知道 CPU 可被用于执行进程、处理中断，也可处于空闲状态（空闲状态分两种，一种是活动进程等待磁盘 IO 导致 CPU 空闲，另一种是完全空闲），其中

(1) sys、usr 字段指示 CPU 被用于处理进程时，进程在内核态、用户态所占 CPU 的时间比例

(2) irq 字段指示 CPU 被用于处理中断的时间比例

(3) idle 字段指示 CPU 处在完全空闲状态的时间比例

(4) wait 字段指示 CPU 处在“进程等待磁盘 IO 导致 CPU 空闲”状态的时间比例

CPU 列各个字段指示值相加结果为 N00%，其中 N 为 cpu 核数。其中

cpu 列：该列显示某一核 cpu 的使用情况，各字段含义可参照 CPU 列，各字段值相加结果为 100%

CPL 列：该列显示 CPU 负载情况

(1) avg1、avg5 和 avg15 字段：过去 1 分钟、5 分钟和 15 分钟内运行队列中的平均进程数量

(2) csw 字段指示上下文交换次数

(3) intr 字段指示中断发生次数

MEM 列：该列指示内存的使用情况

(1) tot 字段指示物理内存总量

(2) free 字段指示空闲内存的大小

(3) cache 字段指示用于页缓存的内存大小

(4) buff 字段指示用于文件缓存的内存大小

(5) slab 字段指示系统内核占用的内存大小

SWP 列：该列指示交换空间的使用情况

(1) tot 字段指示交换区总量

(2) free 字段指示空闲交换空间大小

PAG 列：该列指示虚拟内存分页情况

swin、swout 字段：换入和换出内存页数

DSK 列：该列指示磁盘使用情况，每一个磁盘设备对应一列，如果有 sdb 设备，那么增多一列 DSK 信息

- (1) sda 字段：磁盘设备标识
- (2) busy 字段：磁盘忙时比例
- (3) read、write 字段：读、写请求数量

NET 列：多列 NET 展示了网络状况，包括传输层（TCP 和 UDP）、IP 层以及各活动的网口信息

- (1) XXXi 字段指示各层或活动网口收包数目
- (2) XXXo 字段指示各层或活动网口发包数目

vmstat

前面介绍过，vmstat 命令显示实时的和平均的统计，覆盖 CPU、内存、I/O 等内容，例如内存情况，不仅显示物理内存，也统计虚拟内存。

Linux 上可以用 vmstat 输出中的 free 列监控页面交换，也可以用其他方法例如 top 命令或 /proc/meminfo 文件来监控。监控 vmstat 中的 si 和 so，它们分别表示内存页面换入和换出的量。采用 vmstat -s 命令可以打印出内存使用情况，如代码清单 6-17 所示。

代码清单 6-17 vmstat 打印内存输出

[root@node1:1	zhoumingyao]# vmstat -s
32830232	total memory
26192804	used memory
9182968	active memory
16001608	inactive memory
6637428	free memory
351692	buffer memory
17368908	swap cache
8191996	total swap
543044	used swap
7648952	free swap
25938502	non-nice user cpu ticks
3032	nice user cpu ticks
4571919	system cpu ticks
5000168909	idle cpu ticks
255937	IO-wait cpu ticks
22	IRQ cpu ticks
69875	softirq cpu ticks
0	stolen cpu ticks
5335015	pages paged in
47355696	pages paged out
257031	pages swapped in
2113723	pages swapped out
1159174941	interrupts
1362537573	CPU context switches
1440209401	boot time
6493772	forks

vmstat 命令总结了系统中所有进程使用的总活动虚拟内存，以及空闲列表上实内存页帧的数量。活动的虚拟内存定义为虚拟内存中实际可以得到的工作段页面的数量。这个数字可能大于机器中的实际页帧数，因为一些活动的虚拟内存页可能已写出到调页空间中。

当确定系统内存是否短缺或者是否需要进行某种内存调整时，在设定的时间间隔内运行 vmstat 命令，并检查结果报告中的 pi 和 po 列。这两列表明了每秒调页空间页面调入的数量和每秒调页空间页面调出的数量。如果这些值经常为非零值，说明可能存在内存瓶颈。偶尔出现的非零值不用在意，因为页面调度是虚拟内存的主要原理。

确定系统的适当 RAM 数量的一种方法是查看 vmstat 命令报告的 avm 的最大值。将该数字乘以 4K 得到字节数，然后将其与系统的 RAM 字节数比较。理想情况下，avm 应该小于总 RAM。如果不是，可能会出现一些虚拟内存页面调度量。有多少页面调度发生取决于两个值之间的差值。记住，虚拟内存的概念是提供给我们寻址大于实内存容量的能力（一些在 RAM 内存中，而另一些在调页空间中）。但是如果虚拟内存远大于实内存，可能造成过度的页面调度，从而导致延时。如果 avm 小于 RAM，那么当 RAM 中填满文件页时就会引起调页空间的页面调度。这种情况下，调整 minperm、maxperm 和 maxclient 的值可以减少调页空间的页面调度量。

Sar 工具

Sar 工具可以被用来对内存和交换空间监控。

例如，每 10 秒采样一次，连续采样 3 次，监控内存分页，命令输出如代码清单 6-18 所示。

代码清单 6-18 Sar 收集内存使用情况

```
[root@node186:3 ~]# sar -r 10 3
```

Linux 2.6.32-504.el6.x86_64 (node186) 2015 年 09 月 17 日 _x86_64_ (24 CPU)							
18 时 50 分 15 秒	kmemfree	kmemused	%memused	kbbuffers	kbcached	kbcommit	%commit
18 时 50 分 25 秒	23797880	9032352	27.51	308252	6845976	2902956	7.08
18 时 50 分 35 秒	23791468	9038764	27.53	308256	6846040	2908948	7.09
18 时 50 分 45 秒	23793928	9036304	27.52	308256	6846088	2902780	7.08
平均时间:	23794425	9035807	27.52	308255	6846035	2904895	7.08

输出项逐一说明如下所示。

kmemfree: 这个值和 free 命令中的 free 值基本一致，所以它不包括 buffer 和 cache 的空间；

kmemused: 这个值和 free 命令中的 used 值基本一致，所以它包括 buffer 和 cache 的空间；

%memused: 这个值是 kmemused 和内存总量(不包括 swap)的一个百分比；

kbbuffers 和 kbcached: 这两个值就是 free 命令中的 buffer 和 cache；

kbcommit: 保证当前系统所需要的内存,即为了确保不溢出而需要的内存(RAM+swap)；

%commit: 这个值是 kbcommit 与内存总量(包括 swap)的一个百分比。

针对内存换页监控代码为 sar -B 10 3，运行后输出如代码清单 6-19 所示。

代码清单 6-19 Sar 针对内存换页监控

```
[root@node186:3 ~]# sar -B 10 3
```


Linux 2.6.32-504.el6.x86_64 (node186) 2015 年 09 月 17 日 _x86_64_ (24 CPU)									
18 时 53 分 35 秒	pgpgin/s	pgpgout/s	fault/s	majflt/s	pgfree/s	pgscank/s	pgscand/s	pgsteal/s	%vmeff
18 时 53 分 45 秒	0.00	37.24	7614.21	0.00	3261.06	0.00	0.00	0.00	0.00
18 时 53 分 55 秒	0.00	35.56	7403.20	0.00	3107.59	0.00	0.00	0.00	0.00
18 时 54 分 05 秒	0.00	59.66	7641.34	0.00	3354.85	0.00	0.00	0.00	0.00
平均时间:	0.00	44.15	7552.82	0.00	3241.08	0.00	0.00	0.00	0.00

输出项说明如下所示。

pgpgin/s: 表示每秒从磁盘或 SWAP 置换到内存的字节数(KB);

pgpgout/s: 表示每秒从内存置换到磁盘或 SWAP 的字节数(KB);

fault/s: 每秒钟系统产生的缺页数,即主缺页与次缺页之和(major + minor);

majflt/s: 每秒钟产生的主缺页数;

pgfree/s: 每秒被放入空闲队列中的页个数;

pgscank/s: 每秒被 kswapd 扫描的页个数;

pgscand/s: 每秒直接被扫描的页个数;

pgsteal/s: 每秒钟从 cache 中被清除来满足内存需要的页个数;

%vmeff: 每秒清除的页 (pgsteal) 占总扫描页 (pgscank+pgscand) 的百分比。

总的来说,如果系统使用的内存量保持稳定,也没有启动新应用,却依然有页面调度,说明系统可能在进行页面交换。值得注意的是,如果系统报告可用内存很少,也没有页面调度,说明系统没有页面交换,只不过系统大部分物理 RAM 都被占用了。同样,如果系统在进行页面调度,但内存充足且没有页面交换,说明有应用在启动。

当物理内存逐渐被耗尽时,系统开始将最近最少使用的内存置换到虚拟内存。当应用需要内存页时,就会发生页面换入。随着页面调度的增加,空闲内存基本不变。换句话说,当系统空闲内存很少时,内存页面换入和换出的速度几乎一样快。在 Linux 系统进行页面交换时, Linux vmstat 可以观察到这种典型模式。

6.1.3 监控磁盘

应用程序为了完成一系列工作,可能需要频繁地操作磁盘,无论是关系型数据库、NoSQL 数据库,还是缓存系统,都需要使用到磁盘。因此,对于磁盘的监控相当重要,其重要性不亚于 CPU、内存。

UID 是执行磁盘操作的用户 id。PID 是执行磁盘操作的进程 id。同一块磁盘块 4153884 上有大量的磁盘访问(例如 1024 字节),这意味着存在优化机会,即相同的信息被访问了多次。应用可以在内存中保留和复用这些数据,而不是每次都用昂贵的磁盘操作反复读取。如果读取的数据不相同,则可以一次性读取更大的数据块从而减少磁盘访问的次数。

从更大范围上来说,如果应用的磁盘 I/O 使用率高,就值得深入分析系统磁盘 I/O 子系统的性能,进一步查看它预期的负载量、磁盘服务时间、寻道时间以及服务 I/O 事件的时间。如果需要改善磁盘使用率,可以使用一些策略。从硬件和操作系统上看,更快的存储设备、文件系统扩展

到多个磁盘、操作系统调优使得可以缓存大量的文件系统数据结构，这些都属于改进磁盘 I/O 使用率的策略方式。从应用角度上来看，任何减少磁盘活动的策略都有所帮助，例如使用带宽的输入输出流以减少读、写操作次数，或在应用中集成缓存的数据结构以减少或消除磁盘交互。缓冲流减少了调用操作系统调用的次数从而降低系统态 CPU 使用率。虽然这不会改善磁盘 I/O 性能，但可以使更多 CPU 周期用于应用的其他部分或者其他运行的应用。JDK 提供了缓冲数据结构，也容易使用，如 `java.io.BufferedInputStream` 和 `java.io.BufferedOutputStream`。

关于磁盘性能，有一个经常被忽视的方法，就是检查磁盘缓存是否开启。有一些系统将磁盘缓存设置为禁用。开启磁盘缓存可以改善严重依赖磁盘 I/O 的应用的性能。然而，如果系统默认设置为禁用磁盘缓存，你应该加以注意，因为一旦开启磁盘缓存，意外的电源故障可能会造成数据损坏。

注意，和前面一样，我们本小节的监控方式演示都会按照 Windows 篇和 Linux 篇，其中 Windows 针对的是 Windows7 操作系统，Linux 针对的是 CentosV6.5 操作系统。

6.1.3.1 磁盘使用率监控工具——Windows 篇

性能计数器（Perfmon）

前面已经针对 Perfmon 的使用方式做了详细的介绍，这里不再赘述。Perfmon 提供了比较全面的系统性能指标，并且能够根据性能管理的要求订制日志内容、制定关键指标偏离时的警报措施。表 6-4 列出了 Perfmon 可以监控磁盘相关的性能对象。

表 6-4 Perfmon 磁盘监控功能表

性能对象	计数器	提供的信息
Physical Disk	% Busy Time	% Busy Time 指磁盘驱动器忙于为读或写入请求提供服务所用的时间的百分比
Physical Disk	Avg. Disk Queue Length	Avg. Disk Queue Length 指读取和写入请求（为所选磁盘在实例间隔中队列的）的平均数
Physical Disk	Current Disk Queue Length	Current Disk Queue Length 指在收集操作数据时在磁盘上未完成的请求的数目。它包括在快照内存时正在为其提供服务中的请求。这是一个即时长度而非一定间隔时间的平均值。多主轴磁盘设备可以一次有多个请求操作，但是其他同时发生的请求为等候服务。这个计数器可能会反映一个暂时的高或低的队列长度，但是如果在磁盘驱动器存在持续负载，可能值会总是很高。请求等待时间与这个队列的长度减去磁盘上的主轴成正比。这个差值应小于 2 才能保持良好的性能
Logical Disk	% Free Space	% Free Space 是所选定的逻辑磁盘驱动器上总的可用空闲空间的百分比
Logical Disk	Free Megabytes	可用的 MB 显示磁盘驱动器上尚未分配的空间

6.1.3.2 磁盘使用率监控工具——Linux 篇

对于有磁盘操作的应用来说，查找性能问题，就应该监控磁盘 I/O。一些应用的核心功能需要大量使用磁盘，例如数据库，几乎所有的应用都会用日志记录重要的状态信息或事件发生时的应

用行为。磁盘 I/O 使用率是理解应用磁盘使用情况最有用的监控数据，Linux 可以使用 iostat 来监控系统的磁盘使用率。

iostat 命令

iostat 用于输出 CPU 和磁盘 I/O 相关的统计信息，我们来看一下 iostat 的最基础输出如代码清单 6-20 所示。

代码清单 6-20 iostat 检测磁盘使用的基本输出

```
[root@node186:2 zhoumingyao]# iostat
Linux 2.6.32-504.el6.x86_64 (node186)  2015 年 09 月 16 日  _x86_64_      (24 CPU)
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.44    0.00    0.40    0.00    0.00    99.16

Device:            tps    Blk_read/s    Blk_wrtn/s    Blk_read    Blk_wrtn
sda                 3.28         20.55         97.22     743218     3515392
sdc                 0.02          0.22          0.01        7954         288
sdb                 0.14         21.52          0.10     778106         3544
```

具体解释一下各个字段的意思。

avg-cpu 段

%user: 在用户级别运行所使用的 CPU 的百分比。

%nice: nice 操作所使用的 CPU 的百分比。

%sys: 在系统级别（kernel）运行所使用 CPU 的百分比。

%iowait: CPU 等待硬件 I/O 时，所占用 CPU 百分比。

%idle: CPU 空闲时间的百分比。

Device 段

tps: 每秒钟发送到的 I/O 请求数。

Blk_read /s: 每秒读取的 block 数。

Blk_wrtn/s: 每秒写入的 block 数。

Blk_read: 读入的 block 总数。

Blk_wrtn: 写入的 block 总数。

如果需要进一步调用更多的个性化输出，我们需要配置 iostat 的参数，参数表如下所示。

-c: 仅显示 CPU 统计信息，与-d 选项互斥。

-d: 仅显示磁盘统计信息.与-c 选项互斥。

-k: 以 K 为单位显示每秒的磁盘请求数,默认单位块。

-p device | ALL 与-x 选项互斥，用于显示块设备及系统分区的统计信息。也可以在-p 后指定一个设备名，如：# iostat -p devidename 或显示所有设备# iostat -p ALL。

-t: 在输出数据时，打印搜集数据的时间。

- V: 打印版本号和帮助信息。
- x: 输出扩展信息。

如果想要每隔两秒输出一一次针对磁盘的监控,可以采用如下命令和输出如代码清单 6-21 所示。

代码清单 6-21 iostat 每隔两秒检测一次

```
[root@node186:2 zhoumingyao]# iostat -d 2
Linux 2.6.32-504.el6.x86_64 (node186)  2015 年 09 月 16 日  _x86_64_ (24 CPU)
Device:            tps    Blk_read/s    Blk_wrtn/s    Blk_read    Blk_wrtn
sda                 3.28         20.35         97.06       743234     3545336
sdc                 0.02          0.22          0.01        7954         288
sdb                 0.14         21.30          0.10      778106       3544
Device:            tps    Blk_read/s    Blk_wrtn/s    Blk_read    Blk_wrtn
sda                 3.50          0.00        96.00         0         192
sdc                 0.00          0.00          0.00         0          0
sdb                 0.00          0.00          0.00         0          0
```

如果想要让系统自动每隔 2 秒执行一次 iostat 命令,显示一次设备统计信息,总共输出 6 次,命令和输出清单如代码清单 6-22 所示。

代码清单 6-22 一共输出 6 次

```
[root@node186:2 zhoumingyao]# iostat -d 2 6
Linux 2.6.32-504.el6.x86_64 (node186)  2015 年 09 月 16 日  _x86_64_ (24 CPU)
Device:            tps    Blk_read/s    Blk_wrtn/s    Blk_read    Blk_wrtn
sda                 3.29         20.24         96.98       743234     3560504
sdc                 0.02          0.22          0.01        7954         288
sdb                 0.14         21.19          0.10      778106       3544
Device:            tps    Blk_read/s    Blk_wrtn/s    Blk_read    Blk_wrtn
sda                 7.00          0.00       200.00         0         400
sdc                 0.00          0.00          0.00         0          0
sdb                 0.00          0.00          0.00         0          0
Device:            tps    Blk_read/s    Blk_wrtn/s    Blk_read    Blk_wrtn
sda                 3.00          0.00        52.00         0         104
sdc                 0.00          0.00          0.00         0          0
sdb                 0.00          0.00          0.00         0          0
Device:            tps    Blk_read/s    Blk_wrtn/s    Blk_read    Blk_wrtn
sda                 4.50          0.00       136.00         0         272
sdc                 0.00          0.00          0.00         0          0
sdb                 0.00          0.00          0.00         0          0
Device:            tps    Blk_read/s    Blk_wrtn/s    Blk_read    Blk_wrtn
sda                 1.00          0.00         8.00         0          16
sdc                 0.00          0.00          0.00         0          0
sdb                 0.00          0.00          0.00         0          0
Device:            tps    Blk_read/s    Blk_wrtn/s    Blk_read    Blk_wrtn
sda                 7.46          0.00       171.14         0         344
sdc                 0.00          0.00          0.00         0          0
sdb                 0.00          0.00          0.00         0          0
```


如果想要系统自动每隔 2 秒显示一次 sda1、sdc1 两个设备的扩展统计信息，共输出 6 次，命令和输出如代码清单 6-23 所示。

代码清单 6-23 每隔 2 秒显示一次 sda1、sdc1 两个设备的扩展统计信息

```
[root@node186:2 zhoumingyao]# iostat -x sda1 sdc1 2 6
Linux 2.6.32-504.el6.x86_64 (node186) 2015 年 09 月 16 日 _x86_64_ (24 CPU)
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.44    0.00   0.39   0.00   0.00  99.16

Device: rrqm/s  wrqm/s     r/s     w/s  rsec/s  wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda1           0.25    9.16    0.32    2.96   20.15   96.95   35.69    0.01   2.78   0.22
0.07
sdc1           0.01    0.00    0.02    0.00    0.18    0.01   12.08    0.00   0.45   0.39
0.00

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.19    0.00   0.10   0.00   0.00  99.71

Device: rrqm/s  wrqm/s     r/s     w/s  rsec/s  wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda1           0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.00   0.00
sdc1           0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.00   0.00

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.67    0.00   0.40   0.00   0.00  98.94

Device: rrqm/s  wrqm/s     r/s     w/s  rsec/s  wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda1           0.00   16.50    0.00    7.50    0.00   192.00   25.60    0.00   0.20   0.13
0.10
sdc1           0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.00   0.00

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.02    0.00   0.04   0.00   0.00  99.94

Device: rrqm/s  wrqm/s     r/s     w/s  rsec/s  wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda1           0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.00   0.00
sdc1           0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.00   0.00

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.71    0.00   0.42   0.00   0.00  98.87

Device: rrqm/s  wrqm/s     r/s     w/s  rsec/s  wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda1           0.00   11.50    0.00    5.00    0.00   132.00   26.40    0.00   0.10   0.10   0.05
sdc1           0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.00   0.00

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.19    0.00   0.13   0.00   0.00  99.69

Device: rrqm/s  wrqm/s     r/s     w/s  rsec/s  wsec/s avgrq-sz avgqu-sz   await  svctm  %util
sda1           0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.00   0.00
sdc1           0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.00   0.00
```

如果想要系统自动每隔 2 秒显示一次 sda1 及上面所有分区的统计信息，一共输出 2 次，则命令及输出如代码清单 6-24 所示。

代码清单 6-24 显示两次 sda1 及上面所有分区的统计信息

```
[root@node186:2 zhoumingyao]# iostat -p sda1 2 2
Linux 2.6.32-504.el6.x86_64 (node186) 2015 年 09 月 16 日 _x86_64_ (24 CPU)
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
```

	0.44	0.00	0.39	0.00	0.00	99.16
Device:	tps	Blk_read/s	Blk_wrtn/s	Blk_read	Blk_wrtn	
avg-cpu:	%user	%nice	%system	%iowait	%steal	%idle
	0.29	0.00	0.19	0.00	0.00	99.52
Device:	tps	Blk_read/s	Blk_wrtn/s	Blk_read	Blk_wrtn	

df 命令

Linux 系统中需要监控磁盘各分区的使用情况，避免由于各种突发情况，造成磁盘空间被消耗殆尽的情况，例如某个分区被 Oracle 的归档日志耗尽，导致后续的日志文件无法归档，这时 ORACLE 数据库就会出现错误。

一般查看磁盘各分区的使用情况可以通过 df 命令来查看，我们采用 df -h 命令可以输出基本的磁盘数据，如代码清单 6-25 所示。

代码清单 6-25 输出基本磁盘数据

```
[root@node186:2 zhoumingyao]# df -h
```

Filesystem	Size	Used	Avail	Use%	Mounted on
/dev/sda1	184G	31G	144G	18%	/
tmpfs	16G	160K	16G	1%	/dev/shm
/dev/sdc1	459G	70M	435G	1%	/data
/dev/sdb2	451G	75G	354G	18%	/home

如果我们想要获取磁盘的使用百分比，可以采用如下 Shell 脚本，这样可以更加直接地拿到输出信息。

代码清单 6-26 获取磁盘的使用百分比

```
[root@node186:2 zhoumingyao]# df -P | grep /dev | awk '{print $5}' | cut -f 1 -d
"%"
18
1
1
18
```

或者也可以采用下面这个命令，输出结果一样。

代码清单 6-27 获取磁盘的使用百分比

```
[root@node186:2 zhoumingyao]# df -P | grep /dev | awk '{print $5}' | sed 's/%//g'
18
1
1
18
```

vmstat 命令

vmstat 命令也可以输出磁盘监控数据，如代码清单 6-28 所示。

代码清单 6-28 vmstat 输出磁盘监控数据

```
[root@node186:2 zhoumingyao]# vmstat 1 10
procs -----memory----- --swap-- ----io---- --system-- -----cpu-----
 r  b   swpd   free   buff  cache   si   so   bi   bo   in   cs  us  sy  id  wa  st
 0  0     0 29585428 136212 1049588    0    0    1    2    0   41  0  0 99  0  0
 2  0     0 29582200 136216 1049616    0    0    0   72 761 675  0  0 99  0  0
```

vmstat 命令运行的输出和磁盘相关的数据解释如下所示。

Swap 列

si: 从磁盘交换到内存的交换页数量，单位：KB/秒；

so: 从内存交换到磁盘的交换页数量，单位：KB/秒。

IO 列

bi: 发送到块设备的块数，单位：块/秒；

bo: 从块设备接收到的块数，单位：块/秒。

Sar 工具

例如，每 10 秒采样一次，连续采样 3 次，报告缓冲区的使用情况，输出如代码清单 6-29 所示。

代码清单 6-29 Sar 命令输出缓冲区使用情况

```
[root@node186:3 ~]# sar -b 10 3
Linux 2.6.32-504.el6.x86_64 (node186)  2015 年 09 月 17 日  _x86_64_      (24 CPU)
18 时 57 分 29 秒      tps      rtps      wtps  bread/s  bwrtn/s
18 时 57 分 39 秒      4.01      0.00      4.01    0.00    88.18
18 时 57 分 49 秒      3.10      0.00      3.10    0.00    79.20
18 时 57 分 59 秒      2.70      0.00      2.70    0.00    73.60
平均时间:              3.27      0.00      3.27    0.00    80.32
```

输出项说明如下所示。

tps: 每秒钟物理设备的 I/O 传输总量；

rtps: 每秒钟从物理设备读入的数据总量；

wtps: 每秒钟向物理设备写入的数据总量；

bread/s: 每秒钟从物理设备读入的数据量，单位为块/s；

bwrtn/s: 每秒钟向物理设备写入的数据量，单位为块/s。

6.1.4 监控网络

不管 Java 应用在哪个操作系统上运行，都需要工具显示该应用所用网络接口的网络使用率。分布式 Java 应用的性能和扩展性受限于网络带宽或网络 I/O 的性能。举例来说，如果发送到系统网络接口硬件的消息量超过了它的处理能力，消息就会进入操作系统的缓冲区，这会导致应用延迟。此外网络上发生的其他状况也会导致延迟。即便用操作系统的内嵌工具，也难以直接识别和监控网络使用率。

单次读写数据量小而网络读写量大的应用会消耗大量的系统态 CPU，产生大量的系统调用。对于这类应用，减少系统态 CPU 的策略是减少网络读写的系统调用。此外，使用非阻塞的 JavaNIO 而不是阻塞的 Java.net.Socket，减少处理请求和发送响应的线程数，也可以改善应用性能。

从非阻塞 Socket 中读取数据的策略是，应用在每次读请求时尽可能多地读取数据。同样，当往 socket 中写数据时，每个写调用应该尽可能多地写。一些 Java NIO 框架包含了这些事件，例如 Grizzly 项目。Java NIO 框架也有助于简化客户端/服务器类型应用的开发。JDK 提供的 Java NIO 只是一种原始实现，很容易导致 Java API 的误用而使应用性能变差，建议使用 Java NIO 框架。关于 NIO 的相关内容可以参见本书其他章节。

我们本小节的监控方式演示都会按照 Windows 篇和 Linux 篇，其中 Windows 针对的是 Windows7 操作系统，Linux 针对的是 CentosV6.5 操作系统。

6.1.4.1 磁盘使用率监控工具——Windows 篇

前面已经针对 Perfmon 的使用方式做了详细的介绍，这里不再赘述。Perfmon 提供了比较全面的系统性能指标，并且能够根据性能管理的要求订制日志内容、制定关键指标偏离时的警报措施。表 6-5 列出了 Perfmon 可以监控网络相关的性能对象。

表 6-5 Perfmon 监控网络

性能对象	计数器	提供的信息
Network Interface	Bytes Total/sec	Bytes Total/sec 是发送和接收字节的速率，包括帧字符在内。
Network Interface	Packets/sec	Packets/sec 为发送和接收数据包包的速率。

Windows 上监控网络使用率，需要知道被监控网络接口的带宽，以及网络接口传递的数据量。

网络接口每秒传递的字节数可以通过性能计数器\Network Interface(*)\Bytes Total/sec 获得。通配符“*”表示报告的是系统所有网络接口的总带宽。可以用命令 typeperf\Network Interface(*)\Bytes Total/sec 查看网络接口名，然后用你打算监控的网络接口替换通配符“*”。例如，假定 typeperf\Network Interface(*)\Bytes Total/sec 报告网络接口为 Intel[R] 82566DM-2 Gigabit Network Connection, isatap.gateway_2wire.net,Local Area Connnection*11，可以得知系统安装的网络接口卡是 Intel 网卡。在 Performance Monitor 里或用 typeperf 命令添加性能计数器时，你可以用 Intel[R] 82566DM-2 Gigabit Network Connection 替换通配符“*”。

除了接口传递的字节数，还必须获得网络接口的带宽。可以通过性能计数器\Network Interface(*)\Current Bandwidth 获得，其中“*”应该用被监控的网络接口替换。

重点需要注意的是，性能计数器 Current Bandwidth 的带宽单位是 bits/s。相比而言，Bytes Total/sec 是 bytes。所以网络使用率的计算公式需要考虑适当的单位，bits/s 或 bytes/s。

下面是两个计算网络使用率的攻势：第一个是 Current Bandwidth 除以 8 变为字节，第二个是 Bytes Total/sec 乘以 8 变为比特位。

```
network utilization % = Bytes Total/sec/(Current BandWidth/8)*100
```

或者

```
network utilization % = (Bytes Total/sec*8)/Current BandWidth*100
```

也可以点击 Task Manager 中的 Networking 页监控 Windows 的网络使用率。

6.1.4.2 磁盘使用率监控工具——Linux 篇

Linux 有 netstat 及 sysstat 这两个可选的工具安装包，这两者都不会报告网络使用率。它们的作用时，都可以提供每秒发送和接受的包数，包括错误和冲突的包。少量冲突是以太网的正常情况，大量错误通常是因为网络接口卡出错、糟糕的线路或是自动协商机制(Auto-Negotiation)出了问题。

Netstat 命令

Netstat 命令用于显示各种网络相关信息，如网络连接，路由表，接口状态(Interface Statistics)，masquerade 连接，多播成员 (Multicast Memberships) 等。

从整体上看，netstat 的输出结果可以分为两个部分。

一个是 Active Internet connections，称为有源 TCP 连接，其中"Recv-Q"和"Send-Q"指%0A 的是接收队列和发送队列。这些数字一般都应该是 0。如果不是则表示软件包正在队列中堆积。这种情况只能在非常少的情况见到。

另一个是 Active UNIX domain sockets，称为有源 UNIX 域套接口（和网络套接字一样，但是只能用于本机通信，性能可以提高一倍）。

Proto 显示连接使用的协议,RefCnt 表示连接到本套接口上的进程号,Types 显示套接口的类型,State 显示套接口当前的状态,Path 表示连接到套接口的其他进程使用的路径名。

我们运行 netstat 命令，不使用任何参数，输出如代码清单 6-30 所示。

代码清单 6-30 NetStat 输出

Active Internet connections (w/o servers)						
Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	
tcp	0	0	node186:mysql	10.17.139.251:52684	ESTABLISHED	
tcp	0	0	node186:mysql	10.17.139.251:52671	ESTABLISHED	
tcp	0	0	node186:mysql	10.17.139.251:52674	ESTABLISHED	
tcp	0	0	node186:microsoft-ds	10.17.129.12:57324	ESTABLISHED	
tcp	0	0	node186:microsoft-ds	10.17.128.128:59226	ESTABLISHED	
tcp	0	0	node186:mysql	10.17.139.251:59847	ESTABLISHED	
tcp	0	0	node186:mysql	10.17.139.251:52678	ESTABLISHED	
Active UNIX domain sockets (w/o servers)						
Proto	RefCnt	Flags	Type	State	I-Node	Path
unix	29	[]	DGRAM		13902	/dev/log
unix	2	[]	DGRAM	14809	@/org/freedesktop/hal/udev_event	
unix	2	[]	DGRAM	10507	@/org/kernel/udev/udev	

Netstat 常见参数如下。

- a (all): 显示所有选项，默认不显示 LISTEN 相关；
- t (tcp): 仅显示 tcp 相关选项；
- u (udp): 仅显示 udp 相关选项；
- n: 拒绝显示别名，能显示数字的全部转化成数字；

- l: 仅列出有在 Listen（监听）的服务状态；
- p: 显示建立相关链接的程序名；
- r: 显示路由信息，路由表；
- e: 显示扩展信息，例如 uid 等；
- s: 按各个协议进行统计；
- c: 每隔一个固定时间，执行该 netstat 命令。

注意，LISTEN 和 LISTENING 的状态只有用-a 或者-l 才能看到。

netstat 能报告一定间隔内接收或发送的包数，也仍然难以知道网络接口是否被充分利用。例如，使用 netstat 命令显示 2500 包/秒通过网络接口卡，但你无法知道网络使用率是 100%还是 1%，只能知道存在网络拥堵，而这也是在你不知道底层网络卡传送速率和包大小的情况下，能够得出的唯一结论。简单来说，很难用 Linux 的 netstat 判断应用的性能是否受网络使用率的限制。

如果需要列出所有的端口，包括监听的和未监听的，命令和输出如代码清单 6-31 所示。

代码清单 6-31 NetStat 列出所有端口

```
[root@node186:3 ~]# netstat -a | more
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp      0      0 *:5904                  *:*                     LISTEN
tcp      0      0 node186:mysql           10.17.139.251:51967     ESTABLISHED
tcp      0      0 node186:mysql           10.17.139.251:52681     ESTABLISHED
tcp      0      0 node186:mysql           10.17.139.251:51950     ESTABLISHED
```

如果只想列出 TCP 端口，命令和输出如代码清单 6-32 所示。

代码清单 6-32 NetStat 只列出 TCP 端口

```
[root@node186:3 ~]# netstat -at
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp      0      0 *:6001                  *:*                     LISTEN
tcp      0      0 *:6004                  *:*                     LISTEN
tcp      0      0 192.168.122.1:domain    *:*                     LISTEN
tcp      0      0 *:ssh                   *:*                     LISTEN
tcp      0      0 localhost:ipp           *:*                     LISTEN
tcp      0      0 localhost:smtp          *:*                     LISTEN
```

如果需要列出所有 UDP 端口，命令和输出如代码清单 6-33 所示。

代码清单 6-33 NetStat 只列出 UDP 端口

```
[root@node186:3 ~]# netstat -au
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
udp      0      0 192.168.122.1:domain    *:*
```


udp	0	0 *:ha-cluster	*:*
udp	0	0 localhost:703	*:*
udp	0	0 *:bootps	*:*
udp	0	0 *:974	*:*
udp	0	0 *:sunrpc	*:*
udp	0	0 *:ipp	*:*

如果想要监听 Socket，命令和运行输出如代码清单 6-34 所示。

代码清单 6-34 NetStat 监听 Socket

```
netstat -l
Active UNIX domain sockets (only servers)
Proto RefCnt Flags   Type       State      I-Node Path
unix  2      [ ACC ] STREAM    LISTENING  22877    @/tmp/.ICE-unix/4248
unix  2      [ ACC ] STREAM    LISTENING  22851    /tmp/ssh-lASCjW4248/agent.4248
unix  2      [ ACC ] STREAM    LISTENING  288381   /var/lib/mysql/mysql.sock
unix  2      [ ACC ] STREAM    LISTENING  19249    @/tmp/.ICE-unix/3724
unix  2      [ ACC ] STREAM    LISTENING  23368    /tmp/keyring-xhngwZ/socket
unix  2      [ ACC ] STREAM    LISTENING  23392    /tmp/keyring-xhngwZ/socket.ssh
unix  2      [ ACC ] STREAM    LISTENING  23394    /tmp/keyring-xhngwZ/socket.pkcs11
unix  2      [ ACC ] STREAM    LISTENING  26874    /tmp/orbit-gdm/linc-13a0-0-59afd77a87fa5
unix  2      [ ACC ] STREAM    LISTENING  26946    /tmp/orbit-gdm/linc-139d-0-794105b88a89b
unix  2      [ ACC ] STREAM    LISTENING  27058    /tmp/orbit-gdm/linc-13a5-0-520f3ac6aa15a
unix  2      [ ACC ] STREAM    LISTENING  27095    /tmp/orbit-gdm/linc-13a7-0-18912d61b50d5
```

ifstat 工具

ifstat 工具是个网络接口监测工具，比较简单看网络流量。简单的执行如下所示，默认 ifstat 不监控回环接口，显示的流量单位是 KB。

代码清单 6-35 ifstat 模式输出

```
#ifstat
      eth0          eth1
KB/s in KB/s out  KB/s in KB/s out
    0.07    0.20    0.00    0.00
    0.07    0.15    0.58    0.00
```

如果想要监控所有网络接口，可以采用下面的命令。

代码清单 6-36 ifstat 监控所有网络接口

```
# ifstat -a
      lo          eth0          eth1
KB/s in KB/s out  KB/s in KB/s out  KB/s in KB/s out
    0.00    0.00    0.28    0.58    0.06    0.06
    0.00    0.00    1.41    1.13    0.00    0.00
    0.61    0.61    0.26    0.23    0.00    0.00
```

iftop 工具

iftop 是一款实时流量监控工具，监控 TCP/IP 连接等，缺点是无报表功能。必须以 root 身份才能运行。

默认是监控第一块网卡的流量：iftop

监控 eth1：iftop -i eth1

直接显示 IP，不进行 DNS 反解析：iftop -n

直接显示连接埠编号，不显示服务名称：iftop -N

显示某个网段进出封包流量：iftop -F 192.168.1.0/24 or 192.168.1.0/255.255.255.0

通过 iftop 的界面很容易找到哪个 ip 在霸占网络流量，这个是 ifstat 做不到的。不过 iftop 的流量显示单位是 Mb,这个 b 是 bit，是位，不是字节，而 ifstat 的 KB，这个 B 就是字节了，byte 是 bit 的 8 倍。初学者容易被误导。

nload 工具

如果你仅仅是想查询当前服务器的带宽，nload 绝对是个很好用的一个工具，功能虽然很单一，但是很强。虽然不能像 iptraf 那样，可针对 IP，协议等条件来查询，可以实时地监控网卡的流量，分输入流量 Incoming 和输出流量 Outgoing 两部分，同时统计当前、平均、最小、最大、总流量的值，并且用动态图形方式表现出来，让你一目了然。

输入 nload 回车即可看到动态流量信息，也可以指定网卡，如 nload eth1。还可以指定是以 K 或 M 来显示流量，如 nload -u M 显示的流量是以 MB 为单位的。

代码清单 6-37 nload 输出

```
Incoming

Curr: 0.00 MByte/s
Avg: 0.00 MByte/s
Min: 0.00 MByte/s
Max: 0.00 MByte/s
Ttl: 560.00 Byte

Outgoing:

Curr: 0.00 MByte/s
Avg: 0.00 MByte/s
Min: 0.00 MByte/s
Max: 0.00 MByte/s
Ttl: 560.00 Byte
```

6.2 监控 JVM 活动

监控通常是指一种在生产、质量评估或者开发环境中实施的带有预防或主动性的活动。当应用于干系人报告性能问题却没有提供足以找到根本原因的线索时，首先会进行性能监控，随后是性能分析。性能监控也有助于找出对应用响应性或吞吐量尚未造成严重影响的潜在问题。

相比而言,性能分析是一种以侵入方式收集运行性能数据的活动,它会影响应用的吞吐量或响应性。性能分析是对性能监控或是对干系人所报问题的回应,关注的范围通常比性能监控更集中。性能分析很少在生产环境中进行,通常是在质量评估、测试或开发环境中,常常是性能监控之后再采取的行动。

与性能监控和性能分析相比,性能调优是一种为改善应用响应性或吞吐量而更改参数(Tunable)、源代码或属性配置的活动。性能调优通常是在性能监控或性能分析之后进行。

生产环境中应该自始至终地监控应用 JVM 端。JVM 是应用软件的重要组成部分,应该像监控应用自身和操作系统那样监控 JVM。分析 JVM 监控数据,可以知道何时需要 JVM 调优。JVM 版本变更、操作系统变更(配置或版本)、应用版本更新,或者在应用输入发生重大变动时,应该考虑 JVM 调优。由于 JVM 的输入变化而影响 JVM 性能情况对于许多 Java 应用来说司空见惯。所以,监控 JVM 非常重要。

JVM 的监控范围包括垃圾收集、JIT 编译以及类加载。

6.2.1 监控垃圾收集目的

在 JVM 编译期和加载器,甚至运行期已经做了大量的调优操作,但是那些都是 JVM 针对 Java 程序所做的通用的、简单的优化,程序在运行时由于运行环境的复杂性、业务逻辑的复杂性,很多 JVM 是无法进行优化处理的,这就需要我们自己在写代码的时候就注意,以便我们的程序在特定的业务场景发挥到最佳性能。

笔者觉得监控 JVM 的垃圾收集非常重要,因为它对应用的吞吐量和延迟有着深刻的影响。现代 JVM,如 HotSpot VM,可以将每次 GC 的数据直接输出成日志文件,以文本方式查看 GC 统计数据,或者用 GUI 监控工具查看。注意,HotSpot VM 报告垃圾收集数据几乎没有什么额外开销,建议在生产环境中使用。

一般来说,垃圾收集分两种,即次要垃圾收集(也称为新生代垃圾收集,以下称为 Minor GC)和主要垃圾收集(以下称为 Full GC)。Minor GC 收集新生代,Full GC 通常会收集整个堆,包括新生代、老年代和永久代,除了将新生代中的活跃对象提升到老年代之外,还会压缩整理老年代和永久代。因而 Full GC 之后,新生代为空,老年代和永久代也已压缩整理并且只有活跃对象。

如果各项参数设置合理,系统没有超时日志出现,GC 频率不高,GC 耗时不高,那么没有必要进行 GC 优化;如果 GC 时间超过 1~3 秒,或者频繁 GC,则必须优化。如果满足下面的指标,则一般不需要进行 GC:

- Minor GC 执行时间不到 50ms;
- Minor GC 执行不频繁,约 10 秒一次;
- Full GC 执行时间不到 1s;
- Full GC 执行频率不算频繁,不低于 10 分钟 1 次。

如前所述,Minor GC 会释放新生代中不可达对象所占的内存。相比而言,HotSpot VM 的 Full GC 会释放新生代、老年代和永久代中不可达对象所占的内存。开启 Minor GC 和 Full GC 有多种方式,例如开启 -XX: +UseParallelGC 或 -XX: +UseParallelOldGC 时,如果关闭 -XX: +ScavengeBeforeFullGC,HotSpot VM 在 Full GC 之前不会进行 Minor GC;如果开启 -XX: +

ScavengeBeforeFullGC, HotSpot VM 在 Full GC 前会先做一次 Minor GC, 分担一部分 Full GC 原本要做的工作, 在这两次独立的 GC 之前 Java 线程有机会得以运行, 从而缩短最大停顿时间, 但也拉长了整体的停顿时间。具体相关内容我们会在第 7 章介绍。

重要的垃圾收集数据包括:

- 当前使用的垃圾收集器;
- Java 堆的大小;
- 新生代和老年代的大小;
- 永久代的大小;
- Minor GC 的持续时间;
- Minor GC 的频率;
- Minor GC 的空间回收量;
- Full GC 的持续时间;
- Full GC 的频率;
- 每个并发垃圾收集周期内的空间回收量;
- 垃圾收集前后 Java 堆的占用量;
- 垃圾收集前后新生代和老年代的占用量;
- 垃圾收集前后永久代的占用量;
- 是否老年代或永久代的占用触发了 Full GC;
- 应用是否显式调用了 System.gc()。

6.2.2 GC 垃圾回收报告分析

我们可以通过 `-verbosegc` 来打印出 GC 的日志。`-verbosegc` 是一个比较重要的启动参数, 记录每次 gc 的日志。与 `-verbosegc` 配合使用的一些常用参数为:

- `-XX:+PrintGCDetails`, 打印 GC 信息, 这是 `-verbosegc` 默认开启的选项;
- `-XX:+PrintGCTimeStamps`, 打印每次 GC 的时间戳;
- `-XX:+PrintHeapAtGC`: 每次 GC 时, 打印堆信息;
- `-XX:+PrintGCDateStamps (from JDK 6 update 4)`: 打印 GC 日期, 适合于长期运行的服务器;
- `-Xloggc:/home/admin/logs/gc.log`: 制定打印信息的记录的日志位置。

每条 `verbosegc` 打印出的 gc 日志, 都类似于下面的格式, 如清单 6-38 所示。

代码清单 6-38 GC 日志格式

```
time [GC [<collector>: <starting occupancy1> -> <ending occupancy1>(total
occupancy1), <pause time1> secs] <starting occupancy3> -> <ending occupancy3>(total
occupancy3), <pause time3> secs]
```


例如代码清单 6-39 所示的日志输出,按照清单 6-36 所示的格式,具体每一列代表的意义如代码清单 6-40 所示。

代码清单 6-39 GC 日志示例

```
0.756:[Full GC (System) 0.756: [CMS: 0K->1696K(204800K),0.0347096 secs] 11488K->1696K
(252608K),[CMS Perm : 10328K->10320K(131072K)], 0.0347949 secs] [Times: user=0.06 sys=0.00,
real=0.05 secs]
1.728:[GC 1.728: [ParNew: 38272K->2323K(47808K), 0.0092276 secs] 39968K->4019K(252608K),
0.0093169 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
2.642:[GC 2.643: [ParNew: 40595K->3685K(47808K), 0.0075343 secs] 42291K->5381K(252608K),
0.0075972 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
4.349:[GC 4.349: [ParNew: 41957K->5024K(47808K), 0.0106558 secs] 43653K->6720K(252608K),
0.0107390 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
5.617:[GC 5.617:[ParNew: 43296K->7006K(47808K), 0.0136826 secs] 44992K->8702K(252608K),
0.0137904 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
7.429:[GC 7.429: [ParNew: 45278K->6723K(47808K), 0.0251993 secs] 46974K->10551K(252608K),
0.0252421 secs]
```

代码清单 6-40 GC 日志格式解释

time (可选): 执行 GC 的时间,需要添加-XX:+PrintGCDateStamps 参数才有;
 collector: Minor gc 使用的收集器的名字;
 starting occupancy1: GC 执行前新生代空间大小;
 ending occupancy1: GC 执行后新生代空间大小;
 total occupancy1: 新生代总大小;
 pause time1: 因为执行 minor GC, Java 应用暂停的时间;
 starting occupancy3: GC 执行前堆区域总大小;
 ending occupancy3: GC 执行后堆区域总大小;
 total occupancy3: 堆区总大小;
 pause time3: Java 应用由于执行堆空间 GC (包括 full GC) 而停止的时间。

根据 6-40 的每个字段意义,清单 6-39 对应的倒数第二条信息翻译过来,如代码清单 6-41 所示。

代码清单 6-41 倒数第二行意义

```
5.617 (时间戳):[GC (Young GC) 5.617 (时间戳):[ParNew (GC 的区域):43296K (垃圾回收前
的大小)->7006K (垃圾回收以后的大小)(47808K)(该区域总大小),0.0136826secs (回收时间)]44992K
(堆区垃圾回收前的大小)->8702K (堆区垃圾回收后的大小)(252608K)(堆区总大小),0.0137904secs
(回收时间)][Times:user=0.03 (GC 用户耗时)sys=0.00 (GC 系统耗时),real=0.02secs (GC 实际耗
时)]
```

从最后一条 GC 记录中我们可以看到 Young GC 回收了 45278-6723=38555KB 的内存,Heap 区通过这次回收总共减少了 46974-10551=36423KB 的内存。38555-36423=2132KB 说明通过该次 Young GC 有 2132KB 的内存被移动到了 Old Gen。

6.2.3 图形化工具

离线分析是为了汇总垃圾收集数据并从中查找重要的数据模式。垃圾收集数据的离线分析方法有多种,例如将数据载入电子表格或者图表工具。

6.2.3.1 GCHisto

GCHisto 是一个离线分析工具，它从文件读入垃圾收集数据，以表格和图形化方式展现，如图 6-15 所示。

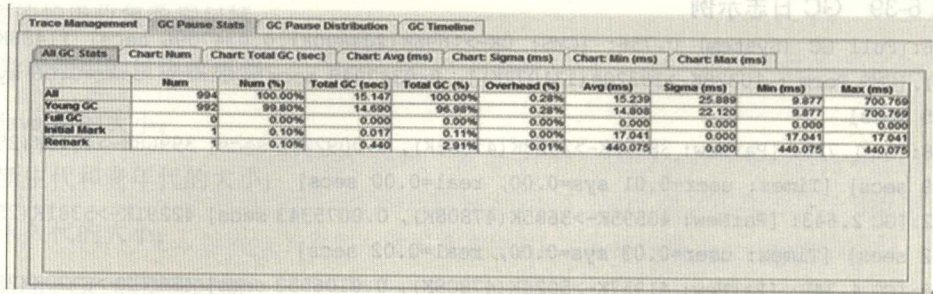


图6-15 GCHisto剪切图

GC Pause Stats TAB 页显示了垃圾收集的次数、开销和持续时间等，它的子选项卡逐项集中展示这些数据。所有引入 Stop-The-World 停顿的垃圾收集，在表中都会占用一行，最上面一行是总计。

垃圾收集的开销（Overhead%）表示垃圾收集调优的程度。作为一般性准则，并发垃圾收集的开销应该小于 10%，也有可能达到 1%-3%。对 Throughput 收集器来说，如果垃圾收集的开销接近 1%，说明垃圾收集器调得很好，3%或更高则说明调优可以改善应用的性能。重要的是理解垃圾收集开销和 Java 堆大小之间的关系，Java 堆越大，降低垃圾收集开销的机会越大。对于给定的 Java 堆大小，通过 JVM 调优才能达到最小的开销。

最长停顿时间（Maximum Pause Time），可以用来评估最差情况下垃圾收集的延时是否满足要求。最长停顿时间超过应用需求时，JVM 可能需要调优，至于是否有必要，则由其超过的程度和超出的停顿时间决定。

默认时，GC Pause Distribution 显示所有垃圾收集停顿的分布，x 轴是垃圾收集的停顿时间，y 轴是停顿的次数。单独查看 Full GC 通常更有用，因为一般来说它的时间最长。只看 Young GC 可以了解停顿时间的变化分布。停顿时间分布广，说明对象分配率和提升率的波动大。如果发现这种情况，你应该查看 GC Timeline，找到垃圾收集活动的峰值。

默认时，GC Timeline 显示整个时间线上所有垃圾收集的停顿。

6.2.3.2 JConsole

JConsole 是一个 JMX(Java Management Extensions)兼容的 GUI 工具，可以连接运行中的 Java5 或者更高版本的 JVM。用 Java5 JVM 启动 Java 应用时，命令行只有添加-Dcom.sun.management.jmxremote，JConsole 才能连接，而 Java6 或更高版本的 JVM 不需要添加此属性。

JConsole 可以以三种方式连接正在运行的 JVM：

- Local: 使用 JConsole 连接一个正在本地系统运行的 JVM，并且执行程序的和运行 JConsole 的需要是同一个用户。JConsole 使用文件系统的授权通过 RMI 连接器连接到平台的 MBean 服务器上。这种从本地连接的监控能力只有 Sun 的 JDK 具有。
- Remote: 使用下面的 URL 通过 RMI 连接器连接到一个 JMX 代理，service:jmx:rmi:///jndi/rmi://hostName:portNum/jmxrmi。hostName 填入主机名称，portNum 为 JMX 代理启动时指

定的端口。JConsole 为建立连接，需要在环境变量中设置 `mx.remote.credentials` 来指定用户名和密码从而进行授权；

- **Advanced:** 使用一个特殊的 URL 连接 JMX 代理。一般情况使用自己定制的连接器而不是 RMI 提供的连接器来连接 JMX 代理，或者是一个使用 JDK1.4 的实现了 JMX 和 JMX Remote 的应用。

JConsole 工具在 `JDK/bin` 目录下，启动 JConsole 后，将自动搜索本机运行的 `jvm` 进程，不需要 `jps` 命令来查询指定。双击其中一个 `jvm` 进程即可开始监控，也可使用“远程进程”来连接远程服务器，如图 6-16 所示。

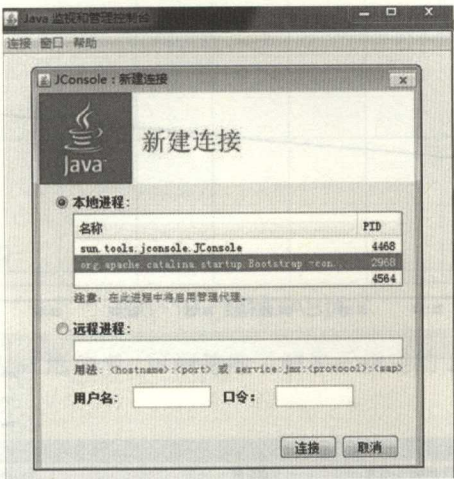


图6-16 登录JConsole

进入 JConsole 主界面，有“概述”、“内存”、“线程”、“类”、“VM 摘要”和“Mbean”等六个选项卡（TAB 页），如图 6-17 所示。

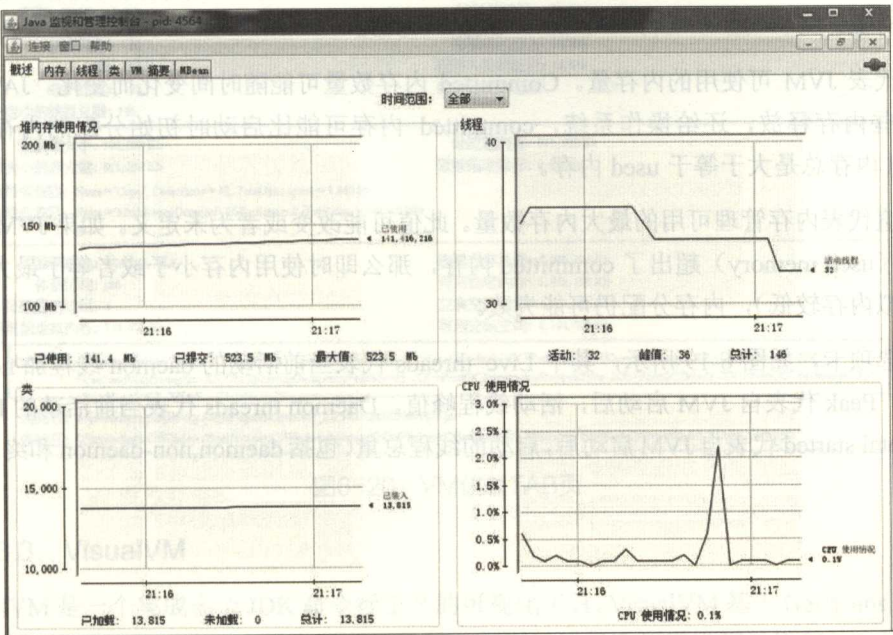


图6-17 JConsole主界面

概要选项卡显示了关于线程使用、内存消耗和 class 加载的一些关键监视信息，以及 JVM 和操作系统的信息。其中 Uptime 代表 JVM 已运行时长，Total compile time 代表花费在即时编译（JIT compilation）中的时间，Process CPU time 代表 JVM 花费的总 CPU 时间。

内存选项卡相当于 jstat 命令，用于监视收集器管理的虚拟机内存（Java 堆和永久代）变化趋势，还可在详细信息栏观察全部 GC 执行的时间及次数，如图 6-18 所示。

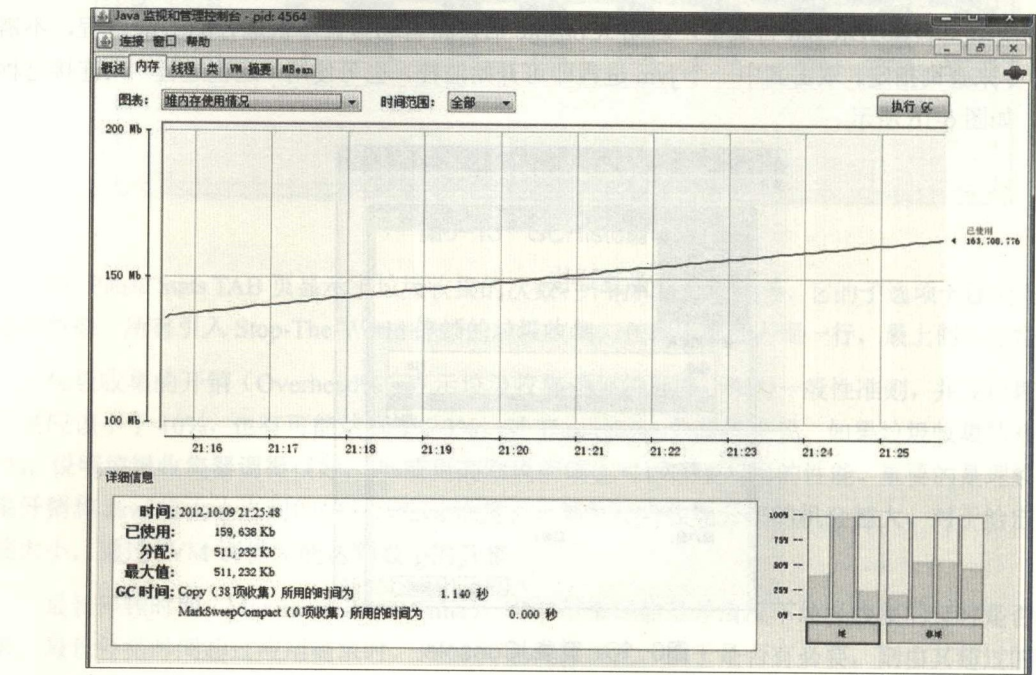


图6-18 内存TAB页

已使用代表当前使用的内存总量。使用的内存总量是指所有的对象占用的内存，包括可达和不可达的对象。

分配代表 JVM 可使用的内存量。Committed 内存数量可能随时间变化而变化。JAVA 虚拟机可能将某些内存释放，还给操作系统，committed 内存可能比启动时初始分配的内存量要少。Committed 内存总是大于等于 used 内存。

最大值代表内存管理可用的最大内存数量。此值可能改变或者为未定义。如果 JVM 试图增加使用内存（used memory）超出了 committed 内存，那么即时使用内存小于或者等于最大内存（比如系统虚拟内存较低），内存分配仍可能失败。

线程选项卡，如图 6-19 所示，其中 Live threads 代表当前活动的 daemon 线程加 non-daemon 线程数量。Peak 代表自 JVM 启动后，活动线程峰值。Daemon threads 代表当前活动的 Daemon 线程数量。Total started 代表自 JVM 启动后，启动的线程总量（包括 daemon、non-daemon 和终止了的）。

息(jps, jinfo), 监视应用程序的 CPU、GC、堆、方法区及线程的信息(jstat、jstack)等。VisualVM 在 JDK/bin 目录下。VisualVM 主界面如图 6-21 所示。

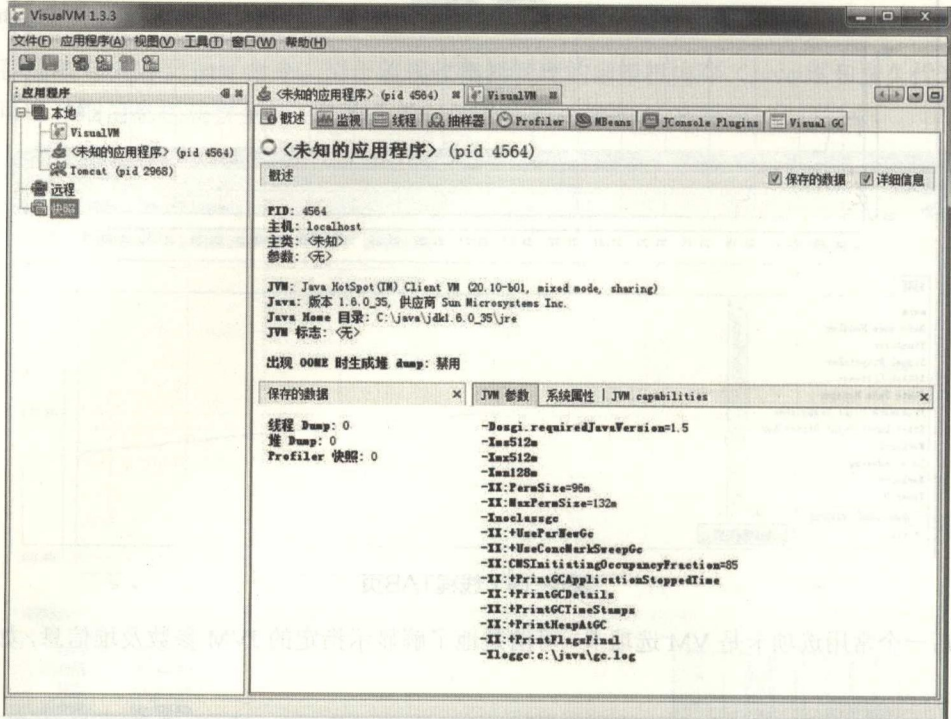


图6-21 VisualVM主界面

使用前我们需要安装插件，如图 6-22 所示。

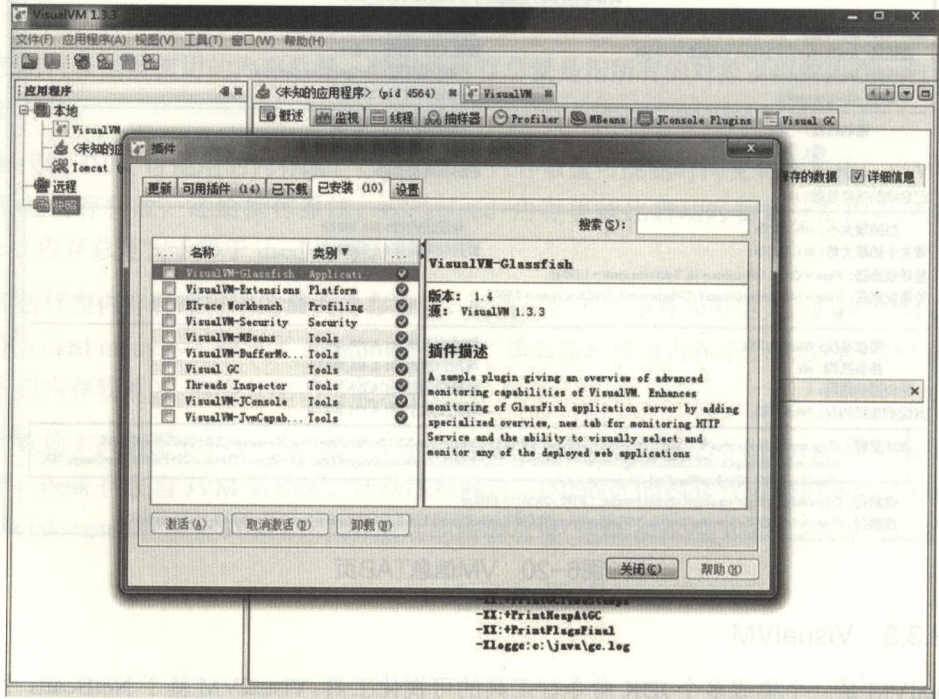


图6-22 VisualVM安装

在 VisualVM 中生成 Dump 文件，如图 6-23 所示。

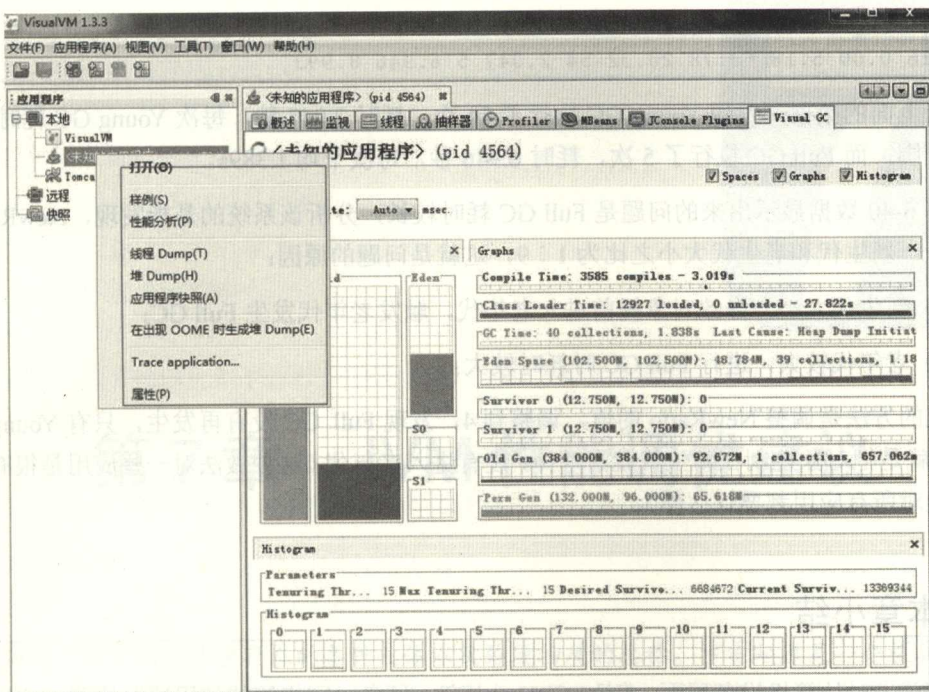


图6-23 VisualVM生成Dump文件

6.2.4 GC 跟踪示例

6.2.4.1 示例一

笔者发现了部分开发测试机器出现异常，异常信息为“java.lang.OutOfMemoryError: GC overhead limit exceeded”。这个异常前半部分可以理解为堆内存不足，后半部分理解为 GC 为了释放很小的空间却耗费了太多的时间。造成这个异常的原因可能有两个，一是可能堆内存设置太小，二是代码中存在死循环或反复创建大对象的可能性存在。笔者首先排除了第 2 个原因，因为这个应用同时是在线上运行的，如果有问题，系统启动的时候就应该出现问题。所以怀疑是这台机器中堆内存设置太小造成了最终的问题。

使用 `ps -ef | grep "java"` 查看，返回结果显示 `-Xms800m`，即该应用的堆区设置只有 800m，而机器内存有 20GB，机器上只运行了这么一个 java 应用，没有其他需要占用内存的地方。另外，这个应用比较大，需要占用的内存也比较多。笔者通过上面的情况判断，只需要改变堆中各区域的大小设置即可，将内存设置为 4GB 后，相关异常没有再次出现。

6.2.4.2 示例二

一个大型服务系统如果经常出现卡顿，那我们就需要考虑 Stop the World 情况的发生可能性了，即是否发生 Full GC 时间太长的情况。

通过 `jstat -gcutil` 命令可以查看 GC 情况，如清单 6-42 所示。

代码清单 6-42 JSTAT 输出 GC 情况

```
S0 S1 E O P YGC YGCT FGC FGCT GCT
12.16 0.00 5.18 63.78 20.32 54 2.047 5 6.946 8.993
```

分析上面的数据，发现 Young GC 执行了 54 次，耗时 2.047 秒，每次 Young GC 耗时 37ms，在正常范围，而 Full GC 执行了 5 次，耗时 6.946 秒，每次平均 1.389s。

清单 6-40 数据显示出来的问题是 Full GC 耗时较长，分析该系统的是指发现，NewRatio=9，也就是说，新生代和老年代大小之比为 1：9，这就是问题的原因：

- (1) 新生代太小，导致对象提前进入老年代，触发老年代发生 Full GC；
- (2) 老年代较大，进行 Full GC 时耗时较大；

优化的方法是调整 NewRatio 的值，调整到 4，发现 Full GC 没有再发生，只有 Young GC 在执行。这就是把对象控制在新生代就清理掉，没有进入老年代（这种做法对一些应用是很有用的，但并不是对所有应用都要这么做）。

6.3 本章小结

本章首先对计算机设备层面，例如 CPU、内存、硬盘、网络等的使用情况检测方式进行了描述，列举了一系列可用的工具及详细用法，接下来介绍了 JVM 监控的一些常用工具及具体使用方法，然后对操作系统层面的一些有用的数据，例如进程、线程、锁竞争、运行时信息等打印方法进行了阐述。

7 chapter

第 7 章 JVM 性能调优建议

宇宙是广袤空间和其中存在的各种天体以及弥漫物质的总称。宇宙起源是一个极其复杂的问题。宇宙是物质世界，它处于不断的运动和发展中。千百年来，科学家们一直在探寻宇宙是什么时候、如何形成的。直到今天，许多科学家认为，宇宙是由大约 137 亿年前发生的一次大爆炸形成的。宇宙内的所存物质和能量都聚集到了一起，并浓缩成很小的体积，温度极高，密度极大，瞬间产生巨大压力，之后发生了大爆炸，这次大爆炸的反应原理被物理学家们称为量子物理。大爆炸使物质四散出去，宇宙空间不断膨胀，温度也相应下降，后来相继出现在宇宙中的所有星系、恒星、行星乃至生命。程序设计优化就好像宇宙起源探索一样，你不了解起源，你就不可能找到正确的方法。

JVM 是 Java 语言可以跨平台、保持高发展的根本，没有了 JVM，Java 语言将失去运行环境。针对 Java 程序的性能优化一定不可能避免针对 JVM 的调优，随着 JVM 的不断发展，我们的应对措施也在不断地跟随、变化。

本章主要介绍和解决以下问题，这也是本书的最核心技术点：

- JVM 的基础架构、生命周期是什么。
- JVM 如何对内部进行管理。
- 垃圾收集器内部原理。
- 常用的 JVM 参数使用及测试结果。
- 如何基于 JVM 对程序调优。

7.1 JVM 相关概念

Java 语言的一大特点就是可以进行自动垃圾回收处理，而无须开发人员过于关注系统资源，例如内存资源的释放情况。自动垃圾收集虽然大大减轻了开发人员的工作量，但是也增加了软件系统的负担。

拥有垃圾收集器可以说是 Java 语言与 C++语言的一项显著区别。在 C++语言中,程序员必须小心谨慎地处理每一项内存分配,且内存使用完后必须手工释放曾经占用的内存空间。当内存释放不够完全时,即存在分配但永不释放的内存块,就会引起内存泄漏,严重时导致程序瘫痪。本节我们先来了解一下 JVM 的内存分配及使用策略,这一点已经在第 2 章重点描述过,所以这里不会过多阐述,接下来会针对字节码、自动内存管理机制进行叙述,这些知识都是为了后面的几个子章节做知识积累的。

7.1.1 内存使用相关概念

7.1.1.1 内存分配意义

内存在计算机世界里占据着至关重要的地位,任何运行时的程序或者数据都需要依靠内存作为存储介质,否则程序将无法正常运行。与 C/C++语言相比,使用 Java 语言编写的程序并不需要显式地为每一个对象都编写对应的内存分配和内存回收等相关函数,这主要是得益于 JVM 的自动内存管理机制,使得 Java 开发人员可以从烦琐的体力劳动中解放出来,只需关注于自身业务即可。

尽管 JVM 的自动内存管理机制大大提升了 Java 开发人员的编程效率,甚至从某种意义上来说还降低了内存泄露和内存溢出的风险,但是如果 Java 开发人员过度依赖于“自动”,那么这将会是一场灾难,最严重的可能性就是会弱化 Java 开发人员在程序出现内存溢出时定位问题和解决问题的能力,事实上也确实如此,我在工作当中实际带领 Java 和 C++两个小组进行开发工作,C++程序员的内存管理意识明显较 Java 程序员强。

JVM 内部定义了多个程序在运行时需要使用到的内存区。JVM 的设计者们之所以会选择将 JVM 的内存结构划分为多个不同的内存区,究其根本原因是因为每一个独立的内存区都拥有各自的用途,都会负责存储各自的数据类型。其中一些内存区的生命周期往往还会和 JVM 的生命周期保持一致,也就是说,会伴随着 JVM 的启动而创建,伴随着 JVM 的退出而销毁。而另一部分内存区则是与线程的生命周期保持一致,会伴随着线程的开始而创建,伴随着线程的消亡而销毁。尽管不同的内存区在存储类型和生命周期上有一定区别,却都拥有一个相同的本质,那就是存储程序的运行时数据。这一节的内容第 2 章已经介绍过,这里不多做介绍,只是让大家能够回忆起来。

JVM 中的内存区可以根据受访权限的不同定义为线程共享和线程私有两大类。

7.1.1.2 线程共享内存区

所谓线程共享指的就是可以允许被所有的线程共享访问的一类内存区,包括堆区、方法区和运行时常量池三个内存区。

■ Java 堆区

Java 堆区在 JVM 启动的时候被创建,并且它在实际的内存空间中可以是不连续的。Java 堆区是一块用于存储对象实例的内存区,同时也是 GC (Garbage Collection, 垃圾收集器) 执行垃圾回收的重点区域。因为 Java 堆区是 GC 的重点回收区域,所以 GC 极有可能会在大内存¹的使用和回收上成为性能瓶颈。为了解决这个问题,JVM 的设计者们开始考虑是否一定需要将对象实例存储

¹ 没有确切的定义,多大叫做大内存,一般来说,以 GB 为单位的对于 JVM 来说都可以称之为大内存。

到 Java 堆区内。

基于 OpenJDK 深度定制的 TaoBaoVM, 其中创新的 GCIH (GC invisible heap)² 技术实现 off-heap, 将生命周期较长的 Java 对象从 Heap 中移至 Heap 之外, 并且 GC 不能管理 GCIH 内部的 Java 对象, 以此达到降低 GC 的回收频率和提升 GC 的回收效率的目的。在某些特殊的应用中有大量生命周期很长的对象, 在应用运行的整个过程中它们都存在, 不需要被 GC 回收。如果这类对象很多, 总体占用内存比例高, 那么他们的存在将给 GC 带来很多不必要的工作负担。假设在淘宝的很多应用中都具有大量的重复对象, 据说这些对象已经占用超过数百 MB 内存, 它们本身在应用提供服务前创建, 在服务过程中永远存在。那么实际在 GC 的收集工作中针对这些对象的所有访问、操作其实都是“无用功”。如果我们把这些对象从 Java 堆内移动到堆外, 那么这些对象所占用的 Java 堆内空间将被释放, GC 的工作量将会降低, 从而每次 Full GC 的时间将会缩短。除此以外, 目前 JVM 间没有很高效的内存/对象共享技术, GCIH 为在 JVM 间共享内存/对象提供了必要的基础。通过这种技术可以将那些移动到 GCIH 内对象在 JVM 间共享, 从而减少内存的总体占用。除此之外, 逃逸分析与栈上分配等优化技术同样也是降低 GC 回收频率和提升 GC 回收效率的一种方式, 这样一来, Java 堆区就不再是 Java 对象内存分配的唯一选择了。

存储在 JVM 中的 Java 对象可以被划分为两类: 一类是生命周期较短的瞬时对象, 这类对象的创建和消亡都非常迅速, 而另外一类对象的生命周期却非常长, 在某些极端的情况下还能够与 JVM 的生命周期保持一致。因此, 对于这些不同生命周期的 Java 对象, 应该采取不同的垃圾收集策略, 分代收集由此诞生。目前几乎所有的 GC 都是用分代收集算法(关于分代收集算法请阅读 7.3.2 节), 所以 Java 堆区如果要进一步细分的话, 还可以划分为新生代 (YoungGen) 和老年代 (OldGen), 其中新生代又可以划分为 Eden 空间、From Survivor 空间和 To Survivor 空间。

既然 Java 堆区用于存储 Java 对象实例, 那么堆的大小在 JVM 启动时就已经设定好了, 大家可以通过选项“-Xmx”和“-Xms”来进行设置。其中选项“-Xms”用于表示堆区的起始内存, 而选项“-Xmx”则用于表示堆区的最大内存。一旦堆区中的内存大小超过“-Xmx”所制定的最大内存时, 将会抛出 OutOfMemoryError 异常。

■ 方法区

方法区和 Java 堆区一样, 同样也是允许被所有的线程共享访问。方法区中存储了每一个 Java 类的结构信息, 比如: 运行时常量池、字段和方法数据、构造函数和普通方法的字节码内容以及类、实例、接口初始化时需要用到的特殊方法等数据。尽管 Java 虚拟机规范对方法区的具体实现方式并没有明确要求, 但是在 HotSpot 中, 方法区仅仅只是逻辑上的独立, 实际上还是被包含在 Java 堆区内, 也就是说, 方法区在物理上属于 Java 堆区的一部分。

方法区在 JVM 启动的时候被创建, 并且它在实际的内存空间中和 Java 堆区一样都可以是不连续的。方法区是一块比较特殊的运行时内存区, 有一些 Java 开发人员更愿意将方法区称之为“永久区 (Permanent Generation)”, 这主要是因为方法区除了可以通过选项“-XX: MaxPermSize”设置内存大小进行动态扩展外, 并不会像 Java 堆区那样频繁地被 GC 执行回收, 甚至还可以显式地制定是否需要在程序运行时回收方法区中的数据, 这就是方法区被大家称之为永久代的原因。但

² 一种将 Java 对象从 Java 堆内移动到堆外, 并且可以在 JVM 间共享这些对象的技术。

是并不表示方法区中的数据永远不会被回收，如果在没有显式要求不对方法区进行内存回收的情况下，GC 的回收目标仅针对方法区中的常量池和类型卸载。

方法区同样也有可能发生内存溢出，一旦方法区中的内存大小超过选项“-XX: MaxPermSize”所指定的最大内存时，将会抛出 `OutOfMemoryError` 异常。

■ 运行时常量池

运行时常量池属于方法区的一部分，一个有效的字节码文件中除了包含类的版本信息、字段、方法以及接口等描述信息外，还包含常量池表（Constant Pool Table），那么运行时常量池就是字节码文件中常量池表的运行时表示形式。运行时常量池中包含多种不同的常量，比如编译器就已经明确的数值字面量、运行期解析后才能够获得的方法或者字段引用。运行时常量池类似于传统编程语言中的符号表（Symbol Table），但是它所包含的数据却比符号表要更丰富一些。

当装载器成功将一个类或者接口装载进 JVM 后，就会创建与之对应的运行时常量池，由于每一个运行时常量池所分配的内存来源于方法区，一旦所需要的内存大小超过方法区所能够提供的最大值时，运行时常量池同样也会抛出 `OutOfMemoryError` 异常。

7.1.1.3 线程私有内存区

和线程共享内存区不同的是，线程私有内存区是不允许被所有线程共享访问的，也就是说，线程私有内存区是只允许被所属的独立线程进行访问的一类内存区，包括 PC 寄存器、Java 栈及本地方法栈等 3 个内存区。

■ PC 寄存器

由于 JVM 是基于栈的架构，所有任何的操作都需要经过入栈和出栈来完成。JVM 中的 PC 寄存器（Program Counter Register）并非是广义上所指的物理寄存器，或者将其翻译为 PC 计数器较为贴切。JVM 中的 PC 寄存器是对物理 PC 寄存器的一种抽象模拟，它是线程私有的，生命周期与线程的生命周期保持一致。如果当前线程所执行的方法是一个 Java 方法，那么 PC 寄存器就会存储正在执行的字节码指令地址，反之如果是 native 方法，这是 PC 寄存器的值就是空（undefined）。

PC 寄存器为什么会被设定为线程私有？我们都知道所谓的多线程在一个特定的时间段内只会执行某一个线程的方法，CPU 会不停地做任务切换，那么为了能够准备地记录各个线程正在执行的当前字节码指令地址，最好的办法自然是为每一个都分配一个 PC 寄存器，这样一来各个线程之间便可以进行独立计算，从而不会出现相互干扰的情况。尽管明确了 PC 寄存器的作用和目的，但是存储字节码指令地址有什么作用呢？JVM 的字节解释器就需要通过改变 PC 寄存器的值来明确下一条应该执行什么样的字节码指令，当然 Java 虚拟机规范并没有明确要求一定要采用这种方式去实现。PC 寄存器是 JVM 的内存区中唯一一个没有明确规定需要抛出 `OutOfMemoryError` 异常的运行时内存区。

■ Java 栈

在 Java 虚拟机规范中，Java 栈也可以被称为 Java 虚拟机栈（Java Virtual Machine Stack），它同 PC 寄存器一样都是线程私有的，并且生命周期与线程的生命周期保持一致。Java 栈用于存储栈帧（Stack Frame），而栈帧中所存储的就是局部变量表、操作数栈，以及方法出口等信息。

Java 堆区中既然存储的是对象实例，那么 Java 栈中的局部变量表就是用于存储各类原始数据类型、对象引用（reference）以及 returnAddress 类型。参考《Java 虚拟机规范 Java SE7 版》的描述来看，returnAddress 类型被定义为 Java 虚拟机内部的原始数据类型，该类型用于表示一条字节码指令的操作码（opcode）。但是 returnAddress 类型在 Java 语言中却不存在相对应的类型，同时自然也无法在运行时更改 returnAddress 类型的值。尽管开发人员无法在程序中直接使用 returnAddress 类型，但在 Java7 之前，该类型却被用于 finally 子句的实现。

Java 栈允许被实现成固定大小的内存或者是可动态扩展的内存大小，如果 Java 栈被设定为固定大小的内存，一旦线程请求分配的栈容量超过 JVM 所允许的最大值时，JVM 将会抛出一个 StackOverflowError 异常，反之抛出一个 OutOfMemoryError 异常。

■ 本地方法栈

本地方法栈（Native Method Stack）用于支持本地方法（native 方法，比如使用 C/C++ 代码编写的方法）的执行，它和 Java 栈的作用类似。Java 虚拟机规范并没有明确要求本地方法栈的具体实现方式，甚至如果 JVM 产品并不打算支持 native 方法，也不依赖于传统栈，则可以无须实现本地方法栈。不过一旦 JVM 中实现有本地方法栈时，那么它将会和 Java 栈一样，允许被实现成固定或者是可动态扩展的内存大小，并且本地方法栈同样也会抛出 StackOverflowError 或者 OutOfMemoryError 异常。

7.1.2 字节码相关知识

7.1.2.1 字节码组织结构

Java 最初诞生的目的就是为在不依赖于特定的物理硬件和操作系统环境下运行，那么也就是说 Java 程序实现跨平台特性的基石其实就是字节码（Byte Code）。Java 之所以能够解决程序的安全性、跨平台移植性等问题，最主要的原因就是 Java 源代码的编译结果并非是本地机器指令，而是字节码。当 Java 源代码成功编译成字节码后，如果想在不同的平台上面运行，则无须再次编译，也就是说 Java 只需一次编译就可处处运行，这就是“Write Once, Run Anywhere”的思想。所以注定了 Java 程序在任何物理硬件和操作系统环境下都能够顺利运行，只要对应的平台装有特定的 Java 运行环境，Java 程序就可以在上面运行，虽然各个平台的 Java 虚拟机内部实现细节不尽相同，但是它们共同执行的字节码内容却是一样的。那么想要让一个 Java 程序正确地运行在 JVM 中，Java 源码就必须要被编译为符合 JVM 规范的字节码。关于规范，开发人员在使用 Java 语言编写一个 Java 程序时需要遵循 Java 语法规范，而将源码编译为字节码的时候又需要符合 JVM 规范。简单地说，前端编译器的主要任务就是负责将符合 Java 语法规范的 Java 代码转换为符合 JVM 规范的字节码文件。

字节码结构组成比较特殊，其内部并不包含任何的分隔符区分段落，所以无论是字节顺序、数量都是有很严格规定的，所以 16 位、32 位、64 位长度的数据都将构造成 2 个、4 个和 8 个 8 位字节单位来表示，多字节数据项总是按照 big-endian 顺序（高位字节载地址最低位，低位字节在地址最高位）来进行存储的，也就是说，一组 8 位字节单位的字节流组成了一个完整的字节码文件。

每一个字节码文件其实都对应着全局唯一的一个类或者接口的定义信息。字节码文件采用的是一种类似于 C 语言结构体的伪结构来描述字节码文件格式。字节码中的每一项都包括类型、名

称以及该项的数量。类型可以是表名，同时也是基本类型，包含在字节码文件中，各项按照严格的顺序进行连续存放，其内部并不包含任何的分隔符区分段落。在这个结构体中只有两种数据结构，分别是无符号数和表。

表是由多个无符号数或者其他表作为数据项构成的复合数据类型，所有表的后缀都是使用“_info”结尾，并且字节码文件实质上也是一张表。尽管使用的是类似于 C 语言的数组语法来表示表中的项，但是无法直接将字节偏移量作为索引对表进行访问，因为表中的每一项的长度都是不固定且可变的。而当描述一个数据结构为数组时，就意味着它由 0 至多个长度固定的项组成，这个时候便可以采用数组索引的方式对其进行访问。每一个字节码文件对应着一个 ClassFile 的结构。

7.1.2.2 javac 编译器

字节码就相当于是一份通用的契约，尽管不同平台上的 Java 虚拟机的实现细节不尽相同，但是它们共同执行的字节码内容却是一样的，这也就是为什么 Java 的设计者们将 Java 的编译结果设定为具有平台通用性的字节码而非本地机器指令的目的所在。

Java 源码的编译结果屏蔽了与底层操作系统和物理硬件相关的一些特性，使得开发人员尽可能地只需关注于自身业务。因为只有这样，体系结构中立、与平台无关等特性才能够真正地构建器来，否则 Java 技术就是一门传统的静态编译型语言，而非今天的动态编译型语言。在 JDK1.0 版本时，Java 程序的运行效率确实不尽如人意，这是因为 Java 当时的解释器非常低效，但这却不代表 Java 语言自身低效，也就是说，程序的运行性能与编译语言其实是没有多大直接关系，真正决定程序运行性能的是编译器。不过在当时，由于技术限制等原因，JVM 中只有解释器，而没有如今先进高效的 JIT 编译器。

Java 源代码的编译结果是字节码，那么肯定需要有一种编译器能够将 Java 源码编译为字节码，承担这个重责的就是配置在“PATH”环境变量中的 javac 编译器。javac 是一种能够将 Java 源码编译为字节码的前端编译器。

HotSpot VM 并没有强制要求前端编译器只能使用 javac 来编译字节码，其实只要编译结果符合 JVM 规范都可以被 JVM 所识别，所以在 Java 的前端编译器领域，除了 javac 之外，还有一种被大家经常用到的前端编译器，那就是内置在 Eclipse 中的 ECJ (Eclipse Compiler for Java) 编译器。和 Javac 的全量式编译不同，ECJ 是一种增量式编译器。在 Eclipse 中，当开发人员编写完代码后，使用“Ctrl+S”快捷键时，ECJ 编译器所采取的编译方案是把未编译部分的源码逐行进行编译，而非每次都全量编译。因此 ECJ 的编译效率会比 javac 更加迅速和高效，当然编译质量和 javac 相比大致还是一样的。这里大家需要注意，前端编译器并不会直接涉及编译优化等方面的技术，而是将这些具体优化细节移交给 HotSpot 的 JIT 编译器负责。

ECJ 不仅是 Eclipse 的默认内置前端编译器，在 Tomcat 中同样也是使用 ECJ 编译器来编译 jsp 文件。由于 ECJ 编译器是采用 GPLv2 的开源协议进行源代码公开，所以，大家可以登录 eclipse 官网下载 ECJ 编译器的源码进行二次开发。

7.1.2.3 编译原理

JVM 并不会只是支持 java 语言，任何语言编写的程序都可以运行在 JVM 中，例如 Scala 语言就可以在 JVM 上运行，前提是源码的编译结果满足并包含 Java 虚拟机的内部指令集、符号表以

及其他的辅助信息，它只要是一个有效的字节码文件，就能够被虚拟机所识别并装载运行。

Javac 编译器在将 Java 源码编译为一个有效的字节码文件过程中经历了 4 个步骤，分别是词法解析、语法解析、语义解析以及生成字节码。

词法解析指的是在 Java 语言中，关键字相信大家都非常熟悉，所谓关键字指的是 Java API 内部预定义的一些字符集合（如 `public`、`private`、`class` 等）。那么词法解析要做的事情就是将 Java 源码中的关键字和标示符等内容转换为符合 Java 语法规则的 Token 序列，然后按照指定的顺序规则进行匹配校验。

语法解析指的是将词法解析后的 Token 序列整合为一棵结构化的抽象语法树，因为我们知道，一个 `try` 语句后面肯定会接上一个 `catch` 或者 `finally` 语句，这就是语法解析步骤。

语义解析指的是由于语法解析器所解析出来的语法数并不能直接应用于生成字节码文件，这是因为这颗语法数相对来说并不完善，所以语义解析的目的就是为了将之前语法解析步骤所产生的语法数扩充得更加完善，后续编译器将会使用语义解析后的语法数来生成字节码。HotSpot 使用了 C++ 以及混合了少量的 C 和汇编代码编写而成的。

词法解析是 javac 编译器执行字节码编译的第一步。在词法解析的过程中，词法解析器最主要的任务就是将 Java 源码中的关键字和标示符等内容转换为符合 Java 语法规则的 Token 序列，然后按照指定的顺序规则进行匹配校验，以便为后续的语法解析步骤做准备。

在 javac 编译器中，词法解析器接口是 `com.sun.tools.javac.parser.Lexer`，它的直接派生实现是位于同包下的 `Scanner` 类，该类的对象实例由 `ScannerFactory` 工厂负责创建。`Scanner` 类的主要任务就是按照单个字符的方式读取 Java 源文件中的关键字和标示符等内容，然后将其转换为符合 Java 语法规则的 Token 序列。负责词法解析工作的并不是 `Scanner` 类，而是 `com.sun.tools.javac.parser.JavacParser` 类，该类的对象实例由 `ParserFactory` 工厂负责创建。也就是说，由 `JavacParser` 类负责控制词法解析时的具体细节，而 `Scanner` 类仅仅只是负责读取源码中的字符集和以及 Token 序列之间的转换任务。

7.1.2.4 符号引用

常量池中主要用于存放字面量（Literal）和符号引用（Symbolic References）两大类数据常量，其访问方式是通过索引来进行访问的，那么符号引用则由 3 种特殊的字符串构成，分别是：全限定名、简单名称和描述符。

在字节码文件中，使用全限定名可以用于描述类或者接口。而类或接口中定义的字段则都会包含一个简单名称和字段描述符。除此之外，定义在类或者接口中的方法，仍然会包含一个简单名称和方法描述符。

字节码文件中包含的所有类或者接口的名称，都是通过全限定名的方式来进行的。`CONSTANT_Class_info` 表中的 `name_index` 是一个指向常量池列表中一个类型为 `CONSTANT_Utf8_info` 常量项的索引，通过这个索引值即可成功获取 `CONSTANT_Utf8_info` 常量项中的全限定名字符串，那么由此可见，类或者接口的名称则被保存在一个类型为 `CONSTANT_Utf8_info` 常量项中。比如一个 `Object` 类型的全限定名为 `java.lang.Object`，但是在字节码文件中，全限定名中的符号 “.” 被符号 “/” 进行了取代，也就是说，`java/lang/Object` 才是 `Object` 类型在字节码文件中的

内部表示形式。那么像 Java API 中的一些其他类型，比如接口 Map 在字节码文件中的全限定名则是 java/util/Map，String 类型在字节码文件中的全限定名则是 java/lang/String。

7.1.2.5 常量池

常量池列表中的常量数并不固定，因此在常量池之前就需要通过一个 2 个字节的常量池计数器来统计常量池列表中到底拥有多少常量项。Java7 种一共包含 14 种类型不尽相同的常量项，但是每个常量项之间的格式都具有一定的通用性，比如每一个常量项中都包含 1 个字节的“tag”项作为开头，它表明了常量项的具体类型和格式，而后面 info[]项的内容则取决于 tag 的类型。

在常量池列表中，CONSTANT_Utf8_info 常量项是一种使用改进过的 UTF-8 编码格式来存储诸如文字字符串、类或者接口的全限定名、字段或者方法的简单名称以及描述符等常量字符串信息。CONSTANT_Utf8_info 常量项的表结构信息如代码清单 7-1 所示。

代码清单 7-1 CONSTANT_Utf8_info 表结构信息

```
CONSTANT_Utf8_info
{
    url tag;
    u2 length;
    u1 bytes[length];
}
```

上述示例代码中，tag 项表明了常量项的具体类型和格式，那么 CONSTANT_Utf8_info 常量项中 tag 项的值则为 CONSTANT_Utf8_info(1)。length 项指明了后续的 bytes[]项的数组长度(字节数)，而 bytes[]项是一个用于存储常量字符串信息的 byte 数组。需要注意的是，改进过的 UTF-8 编码格式与传统的 UTF-8 编码格式存在一定的区别，改进过的 UTF-8 编码格式从‘\u0001’-‘、u007f’之间的字符使用 1 个字节来表示，从‘\u0080’-‘\u07ff’之间的字符使用 2 个字节来表示，而从‘\u0800’-‘\uffff’之间的字符则与传统的 UTF-8 编码格式一样使用 3 个字节来表示。

7.1.2.6 字段表

在字节码文件中，每一个 field_info 项都对应着一个类或者接口中的字段信息，用于表示一个字段的完整信息，比如字段的标示符、访问修饰符（public、private 或 protected）、是类变量还是实例变量（static 修饰符）、是否是常量（final 修饰符）等。在此大家需要注意，由于存储在 field_info 项中的字段信息并不包括声明在方法内部或者代码块内的局部变量，因此多个变量之间的作用域就都是一样的，那么 Java 语法规则必然不允许在一个类或者接口中声明多个具有相同标示符名称的字段。但是和 Java 语法规则相反，字节码文件中却恰恰允许存放多个具有相同标示符名称的字段，唯一的条件就是这些字段之间的描述符不能相同。

field_info 项的表结构信息，如代码清单 7-2 所示。

代码清单 7-2 field_info 项的表结构信息

```
field_info{
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
```



```

u2 attributes_count;
attribute_info attributes[attributes_count];
}

```

上述代码示例中，`access_flag` 项中的值用于存储在类或接口中声明字段时所用到的访问修饰符。

7.1.2.7 方法表

在字节码文件中，每一个 `method_info` 项都对应着一个类或者接口中的方法信息，或者由编译器产生的方法信息（比如：类（接口）初始化方法 `<clinit>()` 和实例初始化方法 `<init>()`），用于表示一个方法的完整信息，比如方法标示符、方法的访问修饰符（`public`、`private` 或 `protected`），方法的返回值类型以及方法的参数信息等。需要注意的是，尽管 Java 语法规则并不允许在一个类或者接口中声明多个方法签名相同的方法，但是和 Java 语法规则相反，字节码文件中却恰恰允许存放多个方法签名相同的方法，唯一的条件就是这些方法之间的返回值不能相同。`method_info` 项的表结构信息如代码清单 7-3 所示。

代码清单 7-3 `method_info` 项的表结构信息

```

method_info{
u2 access_flags;
u2 name_index;
u2 descriptor_index;
us attributes_count;
attribute_info attributes[attributes_count];
}

```

`name_index` 项中的值是一个指向常量池列表中 `CONSTANT_Utf8_info` 常量项的有效索引，通过这个索引值即可成功获取到当前方法的简单名称。而 `descriptor_index` 项中的值同样也是一个指向常量池列表中 `CONSTANT_Utf8_info` 常量项的有效索引，通过这个索引值即可成功获取到当前方法的描述符。`attributes_count` 项指明了后续 `attributes[]` 项的数组长度，`attributes[]` 项中的每一个成员都必须是一个 `attribute_info` 结构的数据项。

7.1.2.8 属性表

属性在字节码文件中几乎随处可见，比如 `ClassFile`、`field_info` 项、`method_info` 项和 `Code_attribute` 项中都是用属性。属性表的通用格式如代码清单 7-4 所示。

代码清单 7-4 `attribute_info` 项的表结构信息

```

attribute_info{
u2 attribute_name_index;
u4 attribute_length;
u1 info[attribute_length];
}

```

`attribute_name_index` 项中的值是一个指向常量池列表的 `CONSTANT_Utf8_info` 常量项的有效索引，通过这个索引值即可成功获取到当前属性的简单名称。`attribute_length` 项指明了后续 `info[]` 项的数组长度（不过并不包括 `attribute_name_index` 项和 `attribute_length` 项在内的起始 6 个字节长度），`info[]` 项中的成员并没有一个明确的要求一定要符合某种结构的数据项，也就是说，`info[]` 项

中的成员结构信息完全是自定义的，该项用于存储属性的数据信息。

7.1.3 自动内存管理

程序内存管理器是一个非常敏感的话题，对于 C/C++ 开发人员而言，他们可以在语法层面随意控制程序中一个对象的生命周期，即能够自由宣布对象诞生，又能够随时宣判对象死亡，因此我们将这种方式称为手动内存管理。尽管手动内存管理非常自由与灵活，但是其弊端同样也非常明显，由于手动内存管理所具备的复杂性，在某些情况下往往会直接或者间接导致程序在运行过程中崩溃，并且一旦出现这种情况，开发人员便会将大把的时间浪费在问题定位上，所以手动内存管理是一把利与弊同样都非常明显的双刃剑。

手动内存管理究竟会发生哪些意外终止程序的正常运行呢？其实综合来看无非就是内存溢出和内存泄漏这两个最主要的原因。内存泄漏也被称为存储渗漏，程序中发生内存泄漏的一个比较常见的场景就是当你打算释放一个链表所引用的所有空间时，却错误地只释放了链表的第一个元素，而剩下的元素尽管已经不再被引用，但是它们却离开了整个程序的控制范围，这样一来，链表中的元素所占用的内存空间将永远无法被释放，这就是内存泄漏。尽管内存泄漏并不会立刻引起程序崩溃，但是一旦发生内存泄漏，程序中的可用内存就会被逐步蚕食，直至耗尽所有内存，最终导致程序崩溃。而内存溢出相对于内存泄漏来说，尽管更容易被理解，但是同样的，内存溢出也是引发程序崩溃的罪魁祸首之一。或许手动内存管理所具备的便捷性和灵活性在某些情况显得非常方便，但是这却法务确保每一个开发人员都能够在创建一个新对象时记得在使用完成之后释放掉其所占用的内存空间。

自动内存管理简单来说就是无须开发人员手动参与内存的分配与回收，这样不仅能够降低内存泄漏和内存溢出的风险，更重要的是自动内存管理机制能够使得开发人员更关注于自身业务。但是对于 Java 开发人员而言，自动内存管理就像是一个黑匣子，如果过度依赖于“自动”，那么这将会是一场灾难，最严重的就会弱化 Java 开发人员在程序出现内存溢出时定位问题和解决问题的能力。所以了解 JVM 的自动内存分配和回收原理就显得非常重要，只有在真正了解 JVM 是如何管理内存后，我们才能够在遇见 `OutOfMemoryError` 时，快速地根据错误异常日志定位问题和解决问题。在高级编程语言中，自动内存管理机制无疑已经是未来的发展趋势，所以除了 Java 采用了自动内存管理机制外，还有像 Lisp、C#、Python、Ruby 等编程语言同样也都采用了自动内存管理机制来实现内存的动态分配和垃圾回收操作。

7.1.3.1 内存分配原理

尽管 Java 对象的内存分配可以选择在堆外进行，但是不可否认这仅仅只是为了降低 GC 回收频率以及提升 GC 回收效率的一种辅助手段，所以 Java 堆区仍然是分配/存储对象实例的主要区域，这一点必然是毋庸置疑的。参考《Java 虚拟机规范 Java SE7 版》的描述来看，JVM 中包含三种引用类型，分别是类类型（class type）、数组类型（array type）和接口类型（interface type），这些引用类型的值则分别由类实例、数组实例以及实现了某个接口的派生类实例负责动态创建，那么 JVM 中究竟是如何为这些类型创建对应的对象实例呢？以创建一个普通的 Java 对象为例，如果是在 Java 语法层面上创建一个对象无非就是使用 `new` 关键字即可，但是在 JVM 中就没有这么简单了。简单来说，当语法层面使用 `new` 关键字创建一个 Java 对象时，JVM 首先会检查这个 `new` 指令的参数能否在常量池中定位到一个类的符号引用，然后检查与这个符号引用相对应的类是否已经成

功经历过加载、解析和初始化等步骤，当类完成装载步骤之后，就已经完全可以确定出创建对象实例时所需要的内存空间大小，接下来 JVM 将会对其进行内存分配，以存储所生成的对象实例。

为新对象分配内存是一件非常严谨和复杂的任务，JVM 的设计者们不仅需要考虑到内存如何分配，在哪里分配等问题，并且由于内存分配算法与内存回收算法密切相关，还需要考虑 GC 执行完内存回收后是否会在内存空间中产生内存碎片。如果内存空间以规整和有序的方式分布，即已用和未用的内存都各自一边，彼此之间维系着一个记录下一条分配起始点的标记指针，当为新对象分配内存时，只需要通过修改指针的便宜量将新对象分配在第一个空闲内存位置上，这种分配方式就叫作指针碰撞（Bump the Pointer），反之则只能使用空闲列表（Free List）执行内存分配。

基于分代的概念，Java 堆区如果还要更进一步细分的话，还可以划分为新生代（YoungGen）和老年代（OldGen），其中新生代内又可以划分为 Eden 空间、From Survivor 空间和 To Survivor 空间。那么对象实例究竟是存储在堆区中的哪一个区域下呢？JVM 的运行时数据区中，堆区和方法区是线程共享区域，任何线程都可以访问到这两个区域中的共享数据，由于对象实例的创建在 JVM 中非常频繁，因此在并发环境下从堆区中划分内存空间是非线程安全的，所以务必需要保证数据操作的原子性。

基于线程安全的考虑，如果一个类在分配内存之前已经成功完成类装载步骤之后，JVM 就会优先选择在 TLAB（Thread Local Allocation，本地线程分配缓冲区）中为对象实例分配内存空间，TLAB 在 Java 堆区中是一块线程私有区域，它包含在 Eden 空间内，除了可以避免一系列的非线程安全问题外，同时还能够提升内存分配的吞吐量，因此我们可以将这种内存分配方式称为快速分配策略。

尽管不是所有的对象实例都能够在 TLAB 中成功分配内存，但 JVM 确实是将 TLAB 作为内存分配的首选，在程序中开发人员可以通过选项“-XX: UseTLAB”设置是否开启 TLAB 空间。TLAB 空间的内存非常小，默认情况下仅占有整个 Eden 空间的 1%，当然我们可以通过选项“-XX: TLABWasteTargetPercent”设置 TLAB 空间所占用 Eden 空间的百分比大小。一旦对象在 TLAB 空间分配内存失败，JVM 就会尝试着通过使用加锁机制确保数据操作的原子性，从而直接在 Eden 空间中分配内存，如果在 Eden 空间中也无法分配内存时，JVM 就会执行 MinorGC，直至最终可以在 Eden 空间中分配内存为止（如果是大对象则直接在老年代中分配）。bytecodeInterpreter.cpp

7.1.3.2 逃逸分析与栈上分配

Java 堆区已经不再是对象内存分配的唯一选择，如果希望降低 GC 的回收频率和提升 GC 的回收频率，那么则可以使用堆外存储技术。目前最常见的堆外存储技术就是利用逃逸分析技术筛选出未发生逃逸的对象，然后避开堆区而直接选择在栈帧中分配内存空间。

逃逸分析（Escape Analysis）是 JVM 在执行性能优化之前的一种分析技术，它的具体目标就是分析出对象的作用域。简单来说，当一个对象被定义在方法体内部之后，它的受访权限仅限于方法体内，一旦其引用被外部成员引用后，这个对象就因此发生了逃逸，反之如果定义在方法体内的对象并没有被任何的外部成员引用，JVM 就会为其在栈帧中分配内存空间。

代码清单 7-5 逃逸分析示例代码

```
public class StackAllocation {
    public StackAllocation obj;
```



```

    public StackAllocation getStackAllocation(){
        //方法返回 StackAllocation 对象实例，发生逃逸
        return null == obj? new StackAllocation() : obj;
    }

    public void setStackAllocation(){
        //为成员变量进行赋值，发生逃逸
        obj = new StackAllocation();
    }

    public void useStackAllocation1(){
        //引用成员变量的值，发生逃逸
        StackAllocation obj = getStackAllocation();
    }

    public void useStackAllocation2(){
        //对象的引用域仅限于方法体内，为发生逃逸
        StackAllocation obj = new StackAllocation();
    }
}

```

由于对象直接在栈上分配内存，因此 GC 就无须执行垃圾回收。栈帧会伴随着方法的调用而创建，伴随着方法的执行结束而销毁，由此可见，栈上分配的对象所占用的内存空间将会随着栈帧的出栈而释放。在 JDK 6u23 版本之后，HotSpot 中默认就已经开启了逃逸分析，如果使用的是较早的版本，开发人员则可以通过选项“-XX: +DoEscapeAnalysis”显式开启逃逸分析，以及通过选项“-XX: +PrintEscapeAnalysis”查看逃逸分析的筛选结果。

7.1.3.3 对象内存布局与 OOP-Klass 模型

当成功对分配后的内存空间执行零值初始化后，JVM 接下来就会对对象进行实例化。在 HotSpot 中，对象实例化操作无非就是初始化对象头和实例数据，而且存储对象实例信息的内存布局也主要由这两个部分构成。先从对象头开始谈起，对象头中主要用于存储 Mark Word 和元数据指针等数据，其中 Mark Word 主要用于存储对象运行时的数据信息，比如 HashCode、GC 分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等。而元数据指针则是用于指向方法区中目标类的类型信息，也就是说通过元数据指针可以准确定位到当前对象的具体目标类型。

除了对象头之外，内存布局的另一部分是实例数据。实例数据主要用于存储定义在当前对象中各种类型的字段信息（包括派生于超类的字段信息），存储在实例对象中的字段顺序除了会与字段在 Java 类中定义的顺序有关外，还会受到 JVM 分配策略参数（FieldsAllocationStyle）的影响，当然开发人员可以通过选项“-XX: FieldsAllocationStyle”设置 JVM 的分配策略参数。HotSpot 默认会按照 longs/doubles、ints、shorts/chars、bytes/booleans、oops 的分配策略顺序进行分配，按照 HotSpot 默认的分配策略后发现，二进制位数相同的字段总是被划分到一起。在满足这个前提的情况下，在超类中定义的变量很有可能会出现在派生类之前。

当大家理解 Java 对象在内存中是如何存储的，接下来我们再来看 JVM 中究竟是如何表达 Java 类以及对象实例的，毕竟 Java 语言只是一个中间语言，运行在 JVM 中还是需要一套完整的底层内部对象表示机制。OOP-Klass 模型就是用于表示 Java 类以及对象实例的一种数据结构，其中 OOP（Ordinary Object Pointer，普通对象指针）用于描述对象的实例信息，而 Klass 则用于描述对象实例的目标类型，也就是说 Klass 其实是一个与 Java 类相对应的 JVM 中的内部对等体，我们也可以

将其称为 C++ 对等体。

OOP 与 Klass 其实是两个相互独立但是却又彼此相互关联的模块,这两个模块均包含在 /openjdk/hotspots/src/share/vm/oops 模块中,那么 OOP-Klass 模型与对象的内存布局之间又有什么关系呢?在 JVM 中对象头就是由 OOP 对象 instanceOopDesc 来表示的(数组类型则用 arrayOopDesc 对象来表示),而对象头中的元数据指针所指向的当前对象的目标类型则是由表示一个 Java 类的对等体。当明确 OOP-Klass 模型的作用后,JVM 可以通过对象引用准确定位到 Java 堆区中的 instanceOopDesc 对象,这样既可成功访问到对象的实例信息,当需要访问目标对象的具体类型时,JVM 则会通过存储在 instanceOopDesc 中的元数据指针定位到存储在方法区中的 instanceKlass 对象上。

7.2 JVM 系统架构

1995 年以来,Java 发生了翻天覆地的变化,出现过许多种 Java 虚拟机,现在越来越多的 Java 应用的性能都能达到要求,这要归功于内建的 JIT 编译器、垃圾收集器,以及不断改进的运行环境(JVM Runtime Environment)。虽然 JVM 有了许多改进,应用的性能和扩展性仍然受到很多人的关注。许多应用的开发需求都增加了性能要求,服务等级协议(Service Level Agreement)中也添加性能条款。随着现代 JVM 性能和扩展性的改善,Java 技术的应用愈加广泛。

若要提高 Java 性能调优的能力,就必须对现代 JVM 有些认识,本书以 HotSpot VM 为例。HotSpot VM 有 3 个主要组件,即 VM 运行时(Runtime)、JIT 编译器(JIT Compiler)以及内存管理器(Memory Manager)。

7.2.1 JVM 的基本架构

早期的 HotSpot VM 是 32 位 JVM,内存地址空间限制为 4GB,此外实现 Java 堆的大小还进一步受限于操作系统。Windows 上 HotSpot VM 最大可用的 Java 堆大约为 1.5GB, Linux 操作系统上最大可用堆大约为 2.5GB 到 3.0GB。实际消耗的最大内存地址空间随给定的 Java 应用和 JVM 版本有所不同。

随着服务器系统内存的不断增大,64 位 HotSpot VM 应运而生。它增大了 Java 堆,使得这些系统可以使用更多内存。虽然 64 位寻址对一些应用有帮助,但 64 位 VM 也带来了性能损失,HotSpot 内部 Java 对象表示(称为普通对象指针,Ordinary Object Pointers,或 oops)的长度从 32 位变成了 64 位,导致 CPU 高速缓存行(CPU Cache Line)中可用的 oops 变少,从而降低了 CPU 缓存的效率。缓存效率的降低常常导致性能比 32 位 JVM 下降 8%~15%。和 OpenJDK 一样,Java6 HotSpot VM 添加了称为压缩指针(Compressed oops,-XX:+UseCompressedOops 开启)的新特性,使得 64 位 Java 的大尺寸堆和 32 位 JVM 的性能都得到了很大提升。压缩指针之所以可以改善性能,是因为它通过对齐(Alignment)、偏移量(Offset)将 64 位指针压缩成 32 位。换言之,性能提高是因为使用了更小更节省空间的压缩指针而不是完整长度的 64 位指针,CPU 缓存使用率由此得以改善,应用程序也能执行得更快。

此外,在一些平台上,例如 Intel 或 AMDx64,64 位 JVM 可以使用更多的 CPU 寄存器,这有

助于程序性能的改善。更多的 CPU 寄存器可以避免寄存器溢出 (Register Spilling)³。当活跃状态 (Live State, 即变量) 数超过 CPU 寄存器, 多出的活跃状态只能存放在内存中时, 就会发生寄存器卸载。寄存器卸载时, 某些活跃状态必须从 CPU 寄存器卸载到内存中。因此, 避免寄存器写是可以让程序执行得更快。

HotSpot 的 Launcher 组件负责维护 JVM 的整个生命周期, 在 Launcher 启动 HotSpot 时, 大部分的功能是通过函数指针指向本地函数, 再由本地函数调用指定模块的具体函数去实现的, 那么当 Launcher 中的启动函数 main() 创建主线程并调用 JavaMain() 函数, 会由 InvocationFunctions 类型的函数指针 CreateJavaJVM 指向本地函数 JNI_CreateJavaJVM() 完成 JVM 的初始化工作, 而在 JNI_CreateJavaJVM() 函数内部却调用了 Threads 模块的 create_vm() 函数来最终完成 JVM 的初始化。

HotSpot 的顶层模块包括:

- adlc 模块中所包含的功能为平台描述文件的编译器。
- asm 模块中所包含的功能为汇编器接口。
- cl 模块中所包含的功能为 client 编译器。
- ci 模块中所包含的功能为动态编译器的公共服务/从动态编译器到 VM 的接口。
- classFile 模块中所包含的功能为类文件的处理, 包括类加载和系统符号表等。
- code 模块中所包含的功能为动态生成的代码的管理。
- compiler 模块中所包含的功能为从 VM 调用动态编译器的接口。

gc 模块由 gc_interface 和 gc_implementation 两部分构成, 其中 gc_interface 模块中所包含的功能为 GC 的接口, 而 gc_implementation 模块中所包含的功能为 GC 的实现, 该模块中还包含了 concurrentMarkSweep、G1、ParallelScavenge、parNew 和 Shared 这 5 个重要的子模块, 其中 concurrentMarkSweep 子模块中所包含的功能为 Concurrent Mark Sweep GC 的实现, G1 子模块所包含的功能为 Garbage-First GC 的实现, parallelScavenge 子模块中所包含的功能为 ParallelScavenge GC 的实现, parNew 子模块中所包含的功能为 ParNewGC 的实现, shared 子模块中所包含的功能为 GC 的共通实现。

interpreter 模块中所包含的功能为内部解释器, 包括模板解释器和 C++ 解释器。

libadt 模块中所包含的功能为一些抽象数据结构。

memory 模块中所包含的功能为内存管理相关代码。

oops 模块中所包含的功能为 server 编译器。

Prims 模块中包含的功能为 HotSpot VM 的对外接口, 包括部分标准库的 native 部分和 JVMTI 实现。

runtime 模块中所包含的功能为运行时支持库, 包括线程管理、编译器调度、锁、反射等。

services 模块中所包含的功能主要是用来支持 JMX 之类的管理功能的接口。

³ 发生在一个程序编辑之间, 在那里有多于寄存器能够保存的活动变量。当一个编译器产生机器代码和有多于这台机器已经寄存的活动变量时, 它不得不从寄存器到内存转换或“溢出”一些变量。

shark 模块中所包含的功能为基于 LLVM 的 JIT 编译器。

utilities 模块中所包含的功能为一些基本的工具类。

7.2.2 JVM 初始化过程

HotSpot VM 运行时环境担当着许多职责,包括命令行选项解析、VM 生命周期管理、类加载、字节码解释、异常处理、同步、线程管理、Java 本地接口、VM 致命错误处理和 C++ (非 Java) 堆管理。

HotSpot VM 运行时系统负责启动和停止 HotSpot VM。启动 HotSpot VM 的组件是启动器。HotSpot VM 有若干个启动器。Unix/Linux 上最常用的是 java, Windows 上是 java 和 javaw, 也可以通过 JNI 接口 JNI_CreateJavaVM 启动内嵌的 JVM。另外还有一个网络启动器 javaws, Web 浏览器用它来启动 applet。javaws 末尾的 ws 通常指的是 web start, 而术语 Java Web Start 即指 javaws。

启动器启动 HotSpot VM 时会执行一系列操作。步骤概述如下:

- (1) 解析命令行选项。启动器会直接处理一些命令行选项,例如-client 或-server,它们决定加载哪个 JIT 编译器,其他参数则传给 HotSpot VM;
- (2) 设置堆的大小和 JIT 编译器。如果命令行没有明确设置堆的大小和 JIT 编译器(client 或 server),启动器则通过自动化有进行设置。自动优化的默认设定因底层系统配置和操作系统而有所不同;
- (3) 设定环境变量,例如 LD_LIBRARY_PATH 和 CLASSPATH;
- (4) 如果命令行有-jar 选项,启动器则从指定 JAR 的 manifest 中查找 Main-Class,否则从命令行读取 Main-Class;
- (5) 使用标准 Java 本地接口 (Java Native Interface, JNI) 方法 JNI_CreateJavaVM 在新创建的线程中创建 HotSpot VM。与后创建的线程相比,初始线程是启动新进程时操作系统内核分配的第一个线程,而新建 HotSpot VM 进程中运行的初始线程也是同样道理。不在初始线程中创建 HotSpot VM,是为了可以对它进行定制,例如 Windows 上更改栈的大小;
- (6) 一旦创建并初始化好 HotSpot VM,就会加载 Java Main-Class,启动器也会从 Java Main-Class 中得到 Java main 方法的参数;
- (7) HotSpot VM 通过 JNI 方法 CallStaticVoidMethod 调用 Java main 方法,并将命令行选项传给它。

至此,HotSpot VM 开始正式执行命令行指定的 Java 程序了。由 Launcher 负责调用 HotSpot 的核心代码对 JVM 执行初始化,以及由它负责维护 JVM 的整个生命周期。

Launcher 是一种用于启动 JVM 进程的启动器,并且可以根据类别划分为两种不同的 Launcher,一种是正式版的启动器,也就是大家在 Windows 平台下经常用到的 java.exe 和 javaw.exe 程序。前者在运行时保留控制台,以及显示程序的输出信息。而后者主要是用于执行 Java 的 GUI 程序,也就是说,使用 javaw.exe 执行 Java 程序时不会显示任何的程序的输出信息。

从严格意义上来说,Launcher 只是一个封装了虚拟机的执行外壳,由它负责状态 JRE 环境和

Windows 平台下的 `jvm.dll` 动态封装库，也就是说，当执行多个 Java 程序时，也就意味着同时启动了多个 JVM 进程。

JVM 的初始化操作其实就是 HotSpot 执行启动的前提条件，并且在初始化过程中还涉及 HotSpot 中一些核心模块的初始，例如初始化 OS 模块、初始化全局数据结构、启动线程、初始化全局模块等。

一旦 Java 程序或者 Java main 方法执行结束，HotSpot VM 就必须检查和清理所有程序或者方法执行过程中生成的未处理异常。此外，方法的退出状态和线程的退出状态也必须返回它们的调用者。调用 Java 本地接口方法 `DetachCurrentThread` 将 Java main 方法与 HotSpot VM 脱离(Detached)。每次 HotSpot VM 调用 `DetachCurrentThread` 时，线程数就会减 1，因此 Java 本地接口知道何时可以安全地关闭 HotSpot VM，并能确保当时 HotSpot VM 中没有正在执行的操作，Java 栈中也没有激活的 Java 帧。

HotSpot VM 启动时 `JNI_CreateJavaVM` 方法将执行以下一系列操作。

- (1) 确保只有一个线程调用这个方法并且确保只创建一个 HotSpot VM 实例。因为 HotSpot VM 创建的静态数据结构无法再次初始化，所以一旦初始化达到某个确定点后，进程空间里就只能有一个 HotSpot VM。HotSpot VM 的启动至此已经无法扭转；
- (2) 检查并确保支持当前的 JNI 版本，初始化垃圾收集日志的输出流；
- (3) 初始化 OS 模块，如随机数生成器 (Random Number Generator)、当前进程 id (Current Process id)、高精度计时器 (High-Resolution Timer)、内存页尺寸 (Memory Page Sizes)、保护页 (Guard Pages)。保护页是不可访问的内存页，用作内存访问区域的边界。例如，操作系统常在线程栈顶压入一个保护页以保证引用不会超出栈的边界；
- (4) 解析传入 `JNI_CreateJavaVM` 的命令行选项，保存以备将来使用；
- (5) 初始化标准的 Java 系统属性，例如 `java.version`、`java.vendor`、`os.name` 等；
- (6) 初始化支持同步、栈、内存和安全点页的模块；
- (7) 加载 `libzip`、`libnpi`、`libjava` 及 `libthread` 等库；
- (8) 初始化并设置信号处理器 (Signal Handler)；
- (9) 初始化线程库；
- (10) 初始化输出流日志记录器 (Logger)；
- (11) 如果用到 Agent 库 (`hprof`、`jdi`)，则初始化并启动；
- (12) 初始化线程状态 (Thread State) 和线程本地存储 (Thread Local Storage)，它们存储了线程私有数据；
- (13) 初始化部分 HotSpot VM 全局数据，例如事件日志 (Event Log)，OS 同步原语、`perfMemory` (性能统计数据内存)，以及 `chunkPool` (内存分配期)；
- (14) 至此，HotSpot VM 可以创建线程了。创建出来的 Java 版 main 线程被关联到当前操作系统的线程，只不过还没有添加到已知线程列表中；

- (15) 初始化并激活 Java 级别的同步;
- (16) 初始化启动类加载器 (Bootclassloader)、代码缓存、解释器、JIT 编译器、JNI、系统词典 (System Dictionary) 及 universe (一种必备的全局数据结构集)。universe 是 HotSpot VM 的一个重要的全局数据结构, 里面包含了一系列与 Java 对象存储相关的重要全局数据结构。universe 这种叫法源自 SmallTalk, 所谓 “universe of objects”, 本质上就是用来存储所有对象的东西的概念。在 HotSpot VM 里, universe 类是一个静态类, 里面是 GCheap、SystemDictionary 等与 Java 对象存储相关的重要结构的引用;
- (17) 现在, 添加 Java 主线程到已知线程列表中。检查 universe 是否正常。创建 HotSpot VM Thread, 它执行 HotSpot VM 所有的关键功能。同时发出适当的 JVMTI 事件, 报告 HotSpot VM 的当前状态;
- (18) 加载和初始化以下 Java 类: java.lang.String、java.lang.System、java.lang.Thread、java.lang.ThreadGroup、java.lang.reflect.Method、java.lang.ref.Finalizer、java.lang.Class 以及余下的 Java 系统类。此时, HotSpot 已经初始化完毕并可使用, 只是功能还不完备;
- (19) 启动 HotSpot VM 的信号处理器线程, 初始化 JIT 编译器并启动 HotSpot 编译代理线程。启动 HotSpot VM 辅助线程 (如监控线程盒统计抽样器)。至此, HotSpot VM 已功能完备;
- (20) 最后, 生成 JNIEnv 对象返回给调用者, HotSpot 则准备相应新的 JNI 请求。

如果 HotSpot VM 启动过程中发生错误, 启动器会调用 DestoryJavaVM 方法关闭 HotSpot VM。如果 HotSpot VM 启动后的执行过程中发生很严重的错误, 也会调用 DestoryJavaVM 方法。

DestoryJavaVM 按以下步骤停止 HotSpot VM。

- (1) 一直等待, 直到只有一个非守护的线程执行, 注意此时 HotSpot VM 仍然可用;
- (2) 调用 java.lang.Shutdown.shutdown() 方法, 它会调用 Java 上的 shutdown 钩子方法, 如果 finalization-on-exit 为 true, 则运行 Java 对象的 finalizer;
- (3) 运行 HotSpot VM 上的 shutdown 钩子 (通过 JVM_OnExit() 注册), 停止以下线程: 性能分析器、统计数据抽样器、监控线程及垃圾收集器线程。发出状态事件通知 JVMTI, 然后关闭 JVMTI、停止信号线程;
- (4) 调用 HotSpot 的 JavaThread::exit() 方法释放 JNI 处理块, 移出保护页, 并将当前线程从已知线程队列中移除。从这时起, HotSpot VM 就无法执行任何 Java 代码了;
- (5) 停止 HotSpot VM 线程, 将遗留的 HotSpot VM 线程带到安全点兵停止 JIT 编译器线程;
- (6) 停止追踪 JNI, HotSpot VM 及 JVMTI 屏障;
- (7) 为那些仍然以本地代码运行的线程设置标记 “vm exited”;
- (8) 删除当前线程;
- (9) 删除或移除所有的输入/输出流, 释放 PerfMemory (性能统计内存) 资源;
- (10) 最后返回到调用者。

从上面的介绍来看，整个过程可以理解为，Launcher 从启动到结束的整个执行过程，成功启动 Launcher 后，首先进入到 Launcher 的启动函数中，这一点和 Java 程序已 iyang，Launcher 的启动函数同样也是 main()。main()函数的主要任务是负责创建运行环境，以及启动一个全新的线程去执行 JVM 的初始化和调用 Java 程序的 main()方法。

当 main()函数成功创建运行后，就会启动一个全新的线程去调用 JavaMain()函数，而 JavaMain()函数的主要任务是负责调用 InitializeJVM()函数。InitializeJVM()函数负责 JVM 初始化的相关工作，但 InitializeJVM()函数本身却并不具备初始化 JVM 的能力，而是由它调用本地函数 JNI_CreateJavaVM()去完成真正意义上的 JVM 初始化。

当 JVM 初始化完成后，Launcher 接着调用 LoadClass()函数和 GetStaticMethodId()函数，分别获取 Java 程序的启动类和启动方法。当执行完这两个步骤后，Launcher 就会调用本地函数 jni_CallStaticMethod()执行 Java 程序的 main()方法。

最后 Launcher 还会调用本地函数 jni_DetachCurrentThread()断开与主线程的链接。当成功与主线程断开连接后，Launcher 就会一直等待程序中所有的非守护线程（non-daemon thread）全部执行结束，然后调用本地函数 jni_DestroyJavaVM()对 JVM 执行销毁。在 JDK1.2 之前，只有主线程才允许对 JVM 执行销毁，而在这之后，非主线程也允许对 JVM 执行销毁。

7.2.3 JVM 架构模型与执行引擎

HotSpot VM 运行时系统解析命令行选项，并据此配置 HotSpot VM。其中一些选项供 HotSpot VM 启动器使用，例如指定选择哪个 JIT 编译器、选择何种垃圾收集器等，还有一些经启动器处理后传给启动的 HotSpot VM，例如制定 Java 堆的大小。

命令行选项主要有 3 类：标准选项（Standard Option）、非标准选项（NonStandard Option）和非稳定选项（Developer Option）。标准选项是 Java Virtual Machine Specification 要求所有 Java 虚拟机都必须实现的选项，它们在发行版之间保持稳定，但也可能在后续的发行版中被废除。非标准选项（以-X 为前缀）不保证、也不强制所有 JVM 实现都必须支持，它可能未经通知就在 Java SDK 发行版之间发生更改。非稳定选项（以-XX 为前缀）通常是为了特定需要而对 JVM 的运行进行校正，并且可能需要有系统配置参数的访问权限。和非标准选项一样，非稳定选项也可能不经通知就在发行版之间发生变动。

命令行选项用于控制 HotSpot VM 的内部变量，每个变量都有类型和默认值。对于内部变量为布尔类型的选项来说，只要在 HotSpot VM 命令行上添加或去掉它就可以控制这些变量。对于带有布尔标记的非稳定选项来说，选项名前的+或-表示 true 或 false，用以开启或关闭特定的 HotSpot VM 特性或参数。例如，-XX: +AggressiveOpts 设置某个 HotSpot 内部布尔变量为 true 以开启额外的性能优化，反之，如果设置为 false，则关闭额外的性能优化。除了布尔标记，还有一类带有附加选项的非稳定选项，形如-XX: OptionName=<N>。几乎所有附加选项为整数的非稳定选项，整数后面都可以接后缀 k,m,g，表示千、百万及十亿。

7.2.4 解释器与 JIT 编译器

HotSpot VM 支持 JIT 编译器的动态优化，可以在 Java 应用运行时制定优化策略，并依据底层系统架构生成高效的本地机器指令。这一点证明 HotSpot VM 架构极富特点且功能强大，可以满足高性能和高扩展性需求。

随着 HotSpot VM 的日渐成熟、运行时环境的不断改进和垃圾收集器的多线程化，即便是最大规模的计算机系统，它也可以充分利用资源，实现高扩展性。

在 HotSpot VM 内部，JIT 编译器（Client 或 Server）和垃圾收集器（Serial、Throughput、CMS 或 G1）都是可插拔的。HotSpot VM 运行时系统为 HotSpot JIT 编译器和垃圾收集器提供服务和通用 API。此外，它还还为 VM 提供启动、线程管理、JNI（Java 本地接口）等基本功能。

7.2.5 类加载机制

任何一个类型在使用之前都必须经历完整的加载、连接和初始化 3 个类加载步骤。一旦这 3 个步骤完成，这个类就可以随时被使用，开发人员可以在程序中访问和调用它的静态类成员信息，比如静态字段、静态方法等，或者使用 `new` 关键字为其创建对象实例。

简单来说，类加载器的主要任务就是根据一个类的全路径名来读取此类的二进制字节流到 JVM 内部，然后转换为一个与目标类对应的 `java.lang.Class` 对象实例。Java 设计者们当初在涉及类加载器的时候，并没有考虑将它绑定在 JVM 内部，这样做的好处是能够更加灵活和动态地执行类加载操作。

JVM 支持两种类型的类加载器，分别为引导类加载器（Bootstrap Classloader）和自定义类加载器（User-Defined Classloader），我们较为常用的是三个类加载器，分别是 `Bootstrap ClassLoader`、`ExtClassLoader`、`AppClassLoader`。

`Bootstrap ClassLoader` 也称之为启动类加载器，它由 C++ 语言编写并嵌套在 JVM 内部，主要负责加载“`JAVA_HOME/lib`”目录中的所有类型，或者由选项“`-Xbootclasspath`”指定路径中的所有类型。`ExtClassLoader` 和 `AppClassLoader` 派生于 `ClassLoader`，并且都是采用 Java 语言编写的，前者主要负责加载“`JAVA_HOME/lib/ext`”扩展目录中的所有类型，后者则主要负责加载 `ClassPath` 目录中的所有类型。

如果当前的类加载器无法满足我们的需求时，便可以在程序中编写自定义类加载器来重新定义类的加载规则，以便实现一些自定义的处理逻辑。在程序中编写一个自定义类加载器是一件非常简单的任务，只需要继承抽象类 `ClassLoader` 并重写 `findClass()` 方法即可。当编写好自定义类加载器后，便可以在程序中调用 `loadClass()` 方法来实现类加载操作。

7.2.5.1 双亲委托模式

Java 虚拟机的设计者们通过一种被称之为双亲委托模型（Parents Delegation Model）的委托机制来约定类加载器的加载机制。按照双亲委托模型的规则，除了启动类加载器之外，程序中每一个类加载器都应该拥有一个超类加载器，比如 `AppClassLoader` 的超类加载器就是 `ExtClassLoader`，而开发人员自己编写的自定义类加载器的超类就是 `AppClassLoader`，那么当一个类加载器接收到一个类加载任务的时候，它并不会立即展开加载，而是将加载任务委派给它的超类加载器去执行，每一层的类加载器都采用相同的方式，直至委派给顶层的启动类加载器为止。如果超类加载器无法加载委托给它的类时，便会将类的加载任务退回给它的下一级类加载器去执行加载。

使用双亲委托模式的优点就是能够有效地确保一个类的全局唯一性，当程序中出现多个相同名字的类名的时候，比如都叫 `hik.michael.Test`，类加载器在执行加载的时候，始终都会只加载其中的某一个类，不会 2 个类都执行加载，如果想通过 `defineClass()` 方法进行显式的加载，JVM 会抛出异常。

注意，由于 Java 虚拟机规范并没有明确要求类加载器的加载机制一定要使用双亲委派模型，只是建议采用这种方式而已。比如在 Tomcat 中，类加载器所采用的加载机制就和传统的双亲委托模型有一定区别，当默认类加载器接收到一个类的加载任务时，首先会由它自动加载，当它加载失败时，才会将类的加载任务委派给它的超类加载器去执行，这也是 Servlet 规范推荐的一种做法。

7.2.5.2 类加载阶段

类的加载阶段就是由类加载器负责根据一个类的全名来读取此类的二进制字节流到 JVM 内部，并存储在运行时内存区中的方法区内，然后将其转换为一个与目标类型对应的 `java.lang.Class` 对象实例（Java 虚拟机规范并没有明确要求一定要存储在 Java 堆区中，因此 HotSpot VM 选择将 Class 对象存储在方法区内），这个 Class 对象在日后就会作为方法区中该类的各种数据的访问入口。而链接阶段要做的事情就是将已经加载到 JVM 中的二进制字节流的类数据信息合并到 JVM 的运行时状态中，然而连接阶段则由验证、准备和解析 3 个阶段构成，其中验证阶段的主要任务就是验证类数据信息是否符合 JVM 规范，是否是一个有效的字节码文件，而验证的内容涵盖了类数据信息的格式验证、语义分析、操作验证等；准备阶段的主要任务就是为类中的所有静态变量分配内存空间，并为其设置一个初始值（由于还没有产生对象，因此实例变量将不在此操作范围内）；而解析阶段的主要任务就是将常量池中所有的符号引用全部转换为直接引用，不过 Java 虚拟机规范并没有明确要求解析阶段一定要按照顺序执行，因此解析阶段可以等到初始化之后再执行。类加载过程中的最后一个阶段就是初始化，在这个阶段中，JVM 会将一个类中所有被 `static` 关键字标识的代码统统执行一遍，如果执行的是静态变量，那么就会使用用户指定的值覆盖掉之前在准备阶段中 JVM 为其设置的初始值，当然如果程序中没有为静态变量显式指定赋值操作，那么所持有的值仍然是之前的初始值；反之如果执行的是 `static` 代码块，那么在初始化阶段中，JVM 将会执行 `static` 代码块中定义的所有操作。

Java 虚拟机规范在类加载和连接的时机上提供了较大的灵活性，但 Java 虚拟机规范却明确规定了类的初始化时机，也就是说，一个类或者接口应该在首次主动使用时执行初始化操作。

7.2.6 虚拟机

7.2.6.1 32 位 VS64 位

相对于传统的 32 位虚拟机，64 位虚拟机所具备的最大优势就是可以访问大内存，32 位虚拟机做的最大可用内存空间被限定在了 4GB，并且 Java 堆区的大小如果是在 Windows 平台下最大只能设置到 1.5GB，而在 Linux 平台下最大也只能设置到 2GB~3GB 的上限，也就是说，Java 堆区的内存大小设置还需要依赖于具体的操作平台。既然 32 位虚拟机无法满足大内存消耗的应用场景，那么 64 位虚拟机的出现则是顺理成章的，64 位虚拟机之所以能够访问大内存，是因为其采用了 64 位的指针架构，这也是寻址访问大内存的关键要素。

在 JDK1.6 Update14 版本之前，64 位虚拟机的综合性能表现实际上是不如 32 位虚拟机的，这主要是因为 OOPS（Ordinary Object Pointers，普通对象指针）从 32 位膨胀到 64 位后，CPU Cache Line 中的可用 OOPS 变少，这样一来就会直接影响并降低 CPU 的缓存使用率，这就是 64 位虚拟机在性能上之所以落后于 32 位虚拟机的主要原因。其次由于部署在 64 位虚拟机上的性能都需要

用到大内存，尤其是互联网项目，经常需要使用多达几十乃至几百 GB 的内存，这对于传统的 32 位虚拟机将无法承载，只能依靠 64 位虚拟机去支撑。但是管理这么大的内存开销对于 GC 来说将会是一场非常严峻的考验，甚至很有可能去导致 GC 在执行内存回收期间消耗更长的时间，同时也意味着工作线程的等待时间将会延长。随着如今 64 位虚拟机的逐渐成熟，指针压缩将会通过对齐补白等操作将 64 位指针压缩为 32 位，以此改善 CPU 缓存使用率达到提升 64 位虚拟机运行性能的目的。

7.2.6.2 JVM 容错示例

Java 的运行时系统也加入了容错机制，其中一个方法就是保证 Java 虚拟机能够支持现役复制 (active replication) 机制。

现役的复制本质上要求复制品服务器像一个确定的、有限状态的机器。Java 虚拟机就是一个非常好的角色。遗憾的是，JVM 并不总是确定的，有很多原因会引起不确定行为，具体原因有如下三点。

- (1) JVM 能够执行本地代码(Native Code)，即 JVM 外部的代码，它向 JVM 提供接口。JVM 对待本地代码就像对待一个黑盒子，也就是说它只能见到接口，不知道调用引起的潜在的不确定性为。因此，为了 JVM 能够主动复制，不应该保证本地代码的行为是确定的。
- (2) 输入数据可能遭受到不确定性。例如，一个能被多个线程操作的共享变量，当允许线程并发访问时，可能被 JVM 转换成不同的实例。为控制这种行为，共享数据至少应该用锁来保护。事实证明，Java 运行时环境没有始终坚持这个原则，尽管其支持多线程。
- (3) 在有故障时，不同的虚拟机产生不同的输出，表明机器被复制了。当这些虚拟机需要返回到相同的状态时，这种不同会引起问题。如果能够假定所有输出都是可以被重放的(bereplayed)或者是可测试的，以至于能够检测输出是否在冲突前产生，问题就能变得简单。为了允许复制品服务器能够决定是否必须重复执行一个操作，这种假设是必需的，

实践表明，把 JVM 变成确定的有限状态机并不简单。一个需要解决的问题是复制服务器可能和主服务器存在冲突。一个可行的方案是让服务器依据“primary-backup”策略工作，在这种工作方案中，一个服务器协调所有需要执行的动作，并一直指示备份服务器做相同的工作，现有很多分布式系统大多都是基于这样的方式来做调度服务器的主备逻辑。

值得注意的是，尽管依照“primary-backup”策略组织了复制服务器，还需要处理现役服务器的复制，即使每个复制品以相同的程序执行相同的操作，我们也不可能完全确保主备服务器不同时崩溃的情况发生。

7.3 垃圾回收机制相关

7.3.1 GC 相关概念

7.3.1.1 GC 的作用

GC (Garbage Collection, 垃圾收集器) 就是 JVM 中自动内存管理机制的具体实现。在 HotSpot 中，GC 的工作任务主要可以划分为两大块，分别是内存的动态分配和垃圾回收。而在内存执行分

配之前，GC 首先会对内存空间进行划分，考虑到 JVM 中存活对象的生命周期会具有两极化，因此应该采取不同的垃圾收集策略，分代收集由此诞生。目前几乎所有的 GC 都是采用分代收集算法执行垃圾回收的，所以 Java 堆区如果还要更进一步细分的话，还可以划分为新生代(YoungGen)和老年代(OldGen)，其中新生代内又可以划分为 Eden 空间、From Survivor 空间和 To Survivor 空间，换句话说，内存空间究竟应该如何划分完全依赖于 GC 的设计。当内存空间划分完成后，GC 就可以为新对象分配内存空间，并区分出存储在内存中的对象哪些是存活的，哪些是已经死亡了的，如果对象已经死亡，那么就可以将其标记为垃圾。为了避免内存溢出，GC 就会释放掉无用对象所占用的内存空间，便于有足够的可用内存空间分配给新的对象实例。

一般来说当内存空间中的内存消耗达到了一定阈值的时候，GC 就会执行垃圾回收，而且回收算法必须非常精确，一定不能造成内存中存活的对象被错误回收掉，也不能造成已经死亡的对象没有被及时地回收掉。而且 GC 执行内存回收的时候应该做到高效，不应该导致应用程序出现长时间的暂停，以及避免产生内存碎片。不过当 GC 执行垃圾回收时，不可避免地会产生一些内存碎片，因为被回收的内存空间极有可能是一些不连续的内存块，这样一来将会导致没有足够的连续可用内存分配给较大的对象，不过可以使用压缩算法消除内存碎片。

淘宝的技术团队对 Java 虚拟机的优化工作其实早已不是停留在简单的参数调整上面，而是充分结合了企业自身的业务特点以及实际的应用场景，在 OpenJDK 的基础之上通过修改大量的 HotSpot 源代码，深度定制了淘宝专属的高性能 Linux 虚拟机 TAOBAOVM。从严格意义上来说，在提升 Java 虚拟机性能的同时，严重依赖于物理 CPU 类型。也就是说，部署有 TAOBAOVM 的服务器中，CPU 全都是清一色的 Intel CPU，且编译手段采用的是 Intel C/CPP Compiler 进行编译，以此对 GC 性能进行提升。除了优化编译效果外，TAOBAOVM 还使用 crc32 指令实现 JVM intrinsic 降低 JNI 的调用开销。

除了在性能优化方面下足了功夫，TAOBAOVM 还在 HotSpot 的基础之上大幅度扩充了一些特定的增强实现，比如创新的 GCIH (GC invisible heap) 技术实现 off-heap，这样一来就可以将生命周期较长的 Java 对象从 heap 中移至 heap 之外，并且 GC 不能管理 GCIH 内部的 Java 对象，这样做最大的好处就是降低了 GC 的回收频率以及提升了 GC 的回收效率，并且 GCIH 中的对象能够在多个 Java 虚拟机进程中实现共享。其他补充技术还有利用 PMU hardware 的 Java profiling tool 和诊断协助功能等。

在许多情况下，GC 不应该成为影响系统性能的瓶颈，可以根据以下 6 点来评估一款 GC 的性能，如下所示。

- 吞吐量：程序的运行时间（程序的运行时间+内存回收的时间）；
- 垃圾收集开销：吞吐量的补数，垃圾收集器所占时间与总时间的比例；
- 暂停时间：执行垃圾收集时，程序的工作线程被暂停的时间；
- 收集频率：相对于应用程序的执行，收集操作发生的频率；
- 堆空间：Java 堆区所占的内存大小；
- 快速：一个对象从诞生到被回收所经历的时间。

7.3.1.2 评价 GC 策略的指标

一个垃圾回收器的应用场景会有它的特定性，我们需要有一些标准来帮助我们评价一个垃圾回收器，我们可以用以下指标评价一个垃圾回收器的好坏。

- 吞吐量：指在应用程序的生命周期内，应用程序所花费的时间和系统总运行时间的比值。系统总运行时间=应用程序耗时+GC 耗时。如果系统运行了 100min，GC 耗时 1min，那么系统的吞吐量就是 $(100-1)/100=99\%$ 。
- 垃圾回收器负载：和吞吐量相反，垃圾回收器负载指来记回收器耗时与系统运行总时间的比值。
- 停顿时间：指垃圾回收器正在运行时，应用程序的暂停时间。对于独占回收器而言，停顿时间可能会比较长。使用并发的回收器时，由于垃圾回收器和应用程序交替运行，程序的停顿时间会变短，但是，由于其效率很可能不如独占垃圾回收器，故系统的吞吐量可能会较低。
- 垃圾回收频率：指垃圾回收器多长时间会运行一次。一般来说，对于固定的应用而言，垃圾回收器的频率应该是越低越好。通常增大堆空间可以有效降低垃圾回收发生的频率，但是可能会增加回收产生的停顿时间。
- 反应时间：指当一个对象称为垃圾后多长时间内，它所占据的内存空间会被释放。
- 堆分配：不同的垃圾回收器对堆内存的分配方式是不同的。一个良好的垃圾收集器应该有一个合理的堆内存区间划分。

7.3.1.3 指针碰撞 (Bump the Pointer) & 空闲列表 (Free List)

在 HotSpot 虚拟机中，使用两种技术加快内存的分配。一个被称为“指针碰撞 (bump-the-pointer)”，另外一个被称为“TLABs (线程本地分配缓冲)”。这两种技术分别是什么呢？

我们知道，Java 是一门面向对象的编程语言，Java 程序运行过程中无时无刻都有对象被创建出来。虚拟机遇到一条 new 指令时，先去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已被加载、解析和初始化过。如果没有，那必须先执行相应的类加载过程。在类加载查通过后，接下来虚拟机将为新生对象分配内存。对象所需内存的大小在类加载完成后便可完全确定（如何确定在下一节对象内存布局时再详细讲解），为对象分配空间的任務具体便等同于一块确定大小的内存从 Java 堆中划分出来，怎么划呢？假设 Java 堆中内存是绝对规整的，所有用过的内存都被放在一边，空闲的内存被放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离，这种分配方式称为“指针碰撞” (Bump The Pointer)。

指针碰撞技术跟踪分配给 Eden 区上最新的对象，该对象将位于 Eden 区的顶部。如果之后有一个对象被创建，只需检查 Eden 区是否有足够大的空间存放该对象。如果空间够用，它将被放置在 Eden 区，存放在空间的顶部。因此，在创建新对象时，只需检查最后被添加对象，看是否还有更多的内存空间允许分配。然而，如果考虑多线程的环境，则是另外一种情况。为了实现多线程环境下，在 Eden 区线程安全的去创建保存对象，那么必须加锁，因此性能会下降。在 HotSpot 虚拟机中 TLABs 能够解决这一问题。它允许每个线程在 Eden 区有自己的一小块私有空间。因为每

一个线程只能访问自己的 TLAB，所以在这个区域甚至可以使用无锁的指针碰撞技术进行内存分配。

如果 Java 堆中的内存并不是规整的，已被使用的内存和空闲的内存相互交错，那就没有办法简单地进行指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式称为“空闲列表”（Free List）。选择哪种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。因此在使用 Serial、ParNew 等带 Compact 过程的收集器时，系统采用的分配算法是指针碰撞，而使用 CMS 这种基于 Mark-Sweep 算法的收集器时（说明一下，CMS 收集器可以通过 UseCMSCompactAtFullCollection 或 CMSFullGCsBeforeCompaction 来整理内存），就通常采用空闲列表。

除了如何划分可用空间之外，还有另外一个需要考虑的问题是对象创建在虚拟机中是非常频繁的行为，即使是仅仅修改一个指针所指向的位置，在并发情况下也并不是线程安全的，可能出现正在给对象 A 分配内存，指针还没来得及修改，对象 B 又同时使用了原来的指针来分配内存。解决这个问题有两个方案，一种是对分配内存空间的动作进行同步——实际上虚拟机是采用 CAS 配上失败重试的方式保证更新操作的原子性；另外一种是把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在 Java 堆中预先分配一小块内存，称为本地线程分配缓冲（TLAB，Thread Local Allocation Buffer），哪个线程要分配内存，就在哪个线程的 TLAB 上分配，只有 TLAB 用完，分配新的 TLAB 时才需要同步锁定。虚拟机是否使用 TLAB，可以通过 -XX:+/-UseTLAB 参数来设定。内存分配完成之后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），如果使用 TLAB 的话，这工作也可以提前至 TLAB 分配时进行。这步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

7.3.1.4 年轻代&老年代

虚拟机给每个对象定义了一个对象年龄（Age）计数器。如果对象在 Eden 出生并经过第一次 Minor GC 后仍然存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并将对象年龄设为 1。对象在 Survivor 区中每熬过一次 Minor GC，年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁）时，就会被晋升到老年代中。对象晋升老年代的年龄阈值，可以通过参数 -XX:MaxTenuringThreshold 来设置。

针对不同年龄段的对象分配原则如下所示。

- （1）对象优先分配在 Eden 区，如果 Eden 区没有足够的空间时，虚拟机执行一次 Minor GC；
- （2）大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在 Eden 区和两个 Survivor 区之间发生大量的内存拷贝（新生代采用复制算法收集内存）；
- （3）长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了 1 次 Minor GC 那么对象会进入 Survivor 区，之后每经过一次 Minor GC 那么对象的年龄加 1，知道达到阈值对象进入老年区；
- （4）动态判断对象的年龄。如果 Survivor 区中相同年龄的所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代；

(5) 空间分配担保。每次进行 Minor GC 时, JVM 会计算 Survivor 区移至老年区的对象的平均大小, 如果这个值大于老年区的剩余值大小则进行一次 Full GC, 如果小于检查 HandlePromotionFailure 设置, 如果 true 则只进行 Monitor GC, 如果 false 则进行 Full GC。

7.3.1.5 Full GC&Minor GC

Minor GC 工作原理如图 7-1 所示。新创建的对象都存放在这里。因为大多数对象很快变得不可达, 所以大多数对象在年轻代中创建, 然后消失。当对象从这块内存区域消失时, 我们说发生了一次 “Minor GC”。

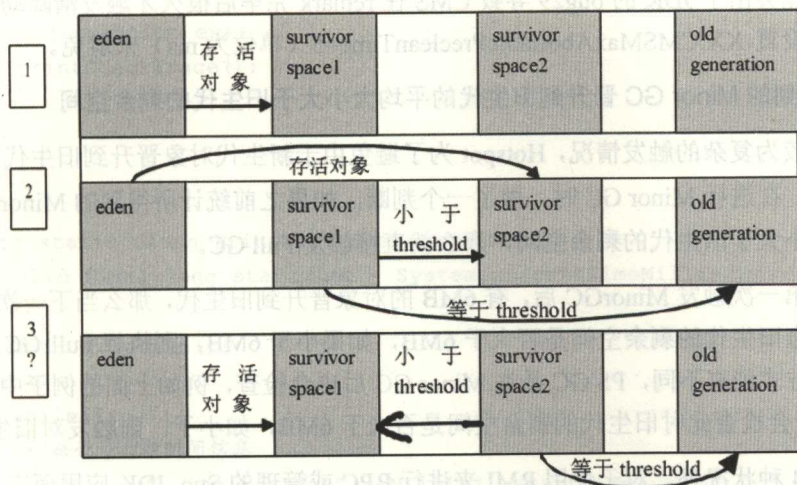


图7-1 Minor GC工作原理图

没有变得不可达, 存活下来的年轻代对象被复制到这里。这块内存区域一般大于年轻代。因为它更大的规模, GC 发生的次数比在年轻代的少。对象从老年代消失时, 我们说 “Major GC” (或 “Full GC”) 发生了。

除了直接调用 System.gc 外, 触发 Full GC 执行的情况有如下四种。

1. 旧生代空间不足

旧生代空间只有在新生代对象转入及创建为大对象、大数组时才会出现不足的现象, 当执行 Full GC 后空间仍然不足, 则抛出如下错误 java.lang.OutOfMemoryError: Java heap space。

为了避免以上两种状况引起的 Full GC, 调优时应尽量做到让对象在 Minor GC 阶段被回收、让对象在新生代多存活一段时间即不要创建过大的对象及数组。

2. Permnet Generation 空间满

Permanet Generation 中存放的为一些 Class 的信息等, 当系统中要加载的类、反射的类和调用的方法较多时, Permanet Generation 可能会被占满, 在未配置为采用 CMS GC 的情况下会执行 Full GC。如果经过 Full GC 仍然回收不了, 那么 JVM 会抛出如下错误信息 java.lang.OutOfMemoryError: PermGen space。

为了避免 Perm Gen 占满造成 Full GC 现象, 可采用的方法为增大 Perm Gen 空间或转为使用 CMS GC。

3. CMS GC 时出现 promotion failed 和 concurrent mode failure

对于采用 CMS 进行旧生代 GC 的程序而言,尤其要注意 GC 日志中是否有 Promotion Failed 和 Concurrent Mode Failure 两种状况,当这两种状况出现时可能会触发 Full GC。Promotion Failed 是在进行 Minor GC 时,Survivor Space 放不下、对象只能放入旧生代,而此时旧生代也放不下造成的;concurrent mode failure 是在执行 CMS GC 的过程中同时对对象要放入旧生代,而此时旧生代空间不足造成的。

应对措施为增大 survivorspace、旧生代空间或调低触发并发 GC 的比率,但在 JDK5.0+、6.0+ 的版本中有可能会由于 JDK 的 bug29 导致 CMS 在 remark 完毕后很久才触发清除动作。对于这种状况,可通过设置-XX:CMSMaxAbortablePrecleanTime=5(单位为 ms)来避免。

4. 统计得到的 Minor GC 晋升到旧生代的平均大小大于旧生代的剩余空间

这是一个较为复杂的触发情况,Hotspot 为了避免由于新生代对象晋升到旧生代导致旧生代空间不足的现象,在进行 Minor GC 时,做了一个判断,如果之前统计所得到的 Minor GC 晋升到旧生代的平均大小大于旧生代的剩余空间,那么就直接触发 Full GC。

例如程序第一次触发 MinorGC 后,有 6MB 的对象晋升到旧生代,那么当下一次 Minor GC 发生时,首先检查旧生代的剩余空间是否大于 6MB,如果小于 6MB,则执行 Full GC。当新生代采用 PS GC 时,方式稍有不同,PS GC 是在 Minor GC 后也会检查,例如上面的例子中第一次 Minor GC 后,PS GC 会检查此时旧生代的剩余空间是否大于 6MB,如小于,则触发对旧生代的回收。

除了以上 4 种状况外,对于使用 RMI 来进行 RPC 或管理的 Sun JDK 应用而言,默认情况下会一小时执行一次 Full GC。可通过在启动时通过- java-Dsun.rmi.dgc.client.gcInterval=3600000 来设置 Full GC 执行的间隔时间或通过-XX:+ DisableExplicitGC 来禁止 RMI 调用 System.gc。

7.3.1.6 Stop the world 案例

垃圾回收时,应用系统会产生一定的停顿。尤其在独占式的垃圾回收器中,整个应用程序会被停止,直到垃圾回收的完成。这种现象称为 Stop-the-World。说得通俗一点,Stop-the-World 意味着 JVM 停止应用程序,而去进行垃圾回收。当 Stop-the-World 发生时,除了进行垃圾回收的线程,其他所有线程都将停止运行。被中断的任务将在 GC 任务完成后恢复执行。GC 调优往往意味着减少 Stop-the-World 的时间。注意,Stop-the-World 会在任何一种 GC 算法中发生。

代码清单 7-6 例子说明垃圾回收对应用程序产生的影响。模拟当生产环境系统出现严重性能问题,即尝试重现了 Stop-the-World 现象。

代码清单 7-6 Stop-the-World 示例

```
import java.util.HashMap;

public class StopWorldDemo {
    public static class MyThread extends Thread{
        HashMap map = new HashMap();
        public void run(){
            try{
                while(true){
```



```

        if(map.size()*512/1024/1024>=400){
            map.clear();//防止内存溢出
            System.out.println("clean map");
        }
        byte[] b1;
        for(int i=0;i<100;i++){
            b1 = new byte[512];//模拟内存占用
            map.put(System.nanoTime(),b1);
        }
    }
} catch (Exception ex){
    ex.printStackTrace();
}
}
}

public static class PrintThread extends Thread{
    public final long starttime = System.currentTimeMillis();

    public void run(){
        try{
            while(true){
                //每毫秒打印时间信息
                long t = System.currentTimeMillis()-starttime;
                System.out.println(t/1000+"."+t%1000);
                Thread.sleep(1000);
            }
        } catch (Exception ex){
            ex.printStackTrace();
        }
    }
}

public static void main(String[] args){
    MyThread t = new MyThread();
    PrintThread p = new PrintThread();
    t.start();
    p.start();
}
}

```

上述代码中定义了两个线程，分别是 MyThread 和 PrintThread。MyThread 不停地申请系统内存，这会迫使进行垃圾回收，PrintThread 则每隔 0.1s 打印出系统启动时间。正常情况下应该 1s 有 10 个输出。

使用参数 -Xmx512M -Xms512M -XX:+UseSerialGC -Xloggc:gc.log -XX:+PrintGCDetails，输出如清单 7-7 所示。

代码清单 7-7 Stop-the-World 示例运行输出

```
0.0
1.14
clean map
2.29
3.309
clean map
4.324
clean map
5.807
clean map
6.978
clean map
7.993
clean map
9.8
clean map
10.148
11.522
clean map
12.755
clean map
```

GC 输出如清单 7-8 所示。

代码清单 7-8 Stop-the-World 示例的 GC 输出

```
0.366: [GC 0.366: [DefNew: 139776K->17472K(157248K), 0.2675607 secs] 139776K->
127626K(506816K), 0.2677332 secs] [Times: user=0.19 sys=0.08, real=0.26 secs]
0.788: [GC 0.788: [DefNew: 157248K->17471K(157248K), 0.3404208 secs] 267402K->
261455K(506816K), 0.3405538 secs] [Times: user=0.31 sys=0.03, real=0.34 secs]
1.241: [GC 1.241: [DefNew: 157247K->157247K(157248K), 0.0000146 secs] 1.241:
[Tenured: 243983K->349567K(349568K), 0.4988513 secs] 401231K->393834K(506816K), [Perm :
380K->380K(12288K)], 0.4990866 secs] [Times: user=0.50 sys=0.00, real=0.50 secs]
1.940: [Full GC 1.940: [Tenured: 349567K->47656K(349568K), 0.1760066 secs]
506815K->47656K(506816K), [Perm : 380K->380K(12288K)], 0.1761751 secs] [Times:
user=0.17 sys=0.00, real=0.17 secs]
2.226: [GC 2.227: [DefNew: 139776K->17472K(157248K), 0.2130968 secs] 187432K->
185248K(506816K), 0.2132255 secs] [Times: user=0.22 sys=0.00, real=0.22 secs]
2.555: [GC 2.555: [DefNew: 157248K->17471K(157248K), 0.2630955 secs] 325024K->
323914K(506816K), 0.2632084 secs] [Times: user=0.27 sys=0.00, real=0.27 secs]
2.922: [GC 2.922: [DefNew: 157247K->157247K(157248K), 0.0000138 secs] 2.922:
[Tenured: 306443K->349567K(349568K), 0.5437268 secs] 463690K->455014K(506816K), [Perm :
380K->380K(12288K)], 0.5439029 secs] [Times: user=0.53 sys=0.00, real=0.53 secs]
3.556: [Full GC 3.556: [Tenured: 349567K->51995K(349568K), 0.1789923 secs] 506815K->
51995K(506816K), [Perm : 380K->379K(12288K)], 0.1791206 secs] [Times: user=0.19
sys=0.00, real=0.19 secs]
3.845: [GC 3.845: [DefNew: 139776K->17471K(157248K), 0.2132299 secs] 191771K->
190545K(506816K), 0.2133538 secs] [Times: user=0.20 sys=0.00, real=0.20 secs]
```


4.174: [GC 4.174: [DefNew: 157247K->17471K(157248K), 0.2663958 secs] 330321K->329202K(506816K), 0.2665229 secs] [Times: user=0.27 sys=0.00, real=0.26 secs]

4.569: [GC 4.569: [DefNew: 157247K->157247K(157248K), 0.0000158 secs] 4.569: [Tenured: 311730K->12917K(349568K), 0.1204363 secs] 468978K->12917K(506816K), [Perm : 379K->379K(12288K)], 0.1206203 secs] [Times: user=0.13 sys=0.00, real=0.13 secs]

4.798: [GC 4.798: [DefNew: 139776K->17471K(157248K), 0.2108548 secs] 152693K->151356K(506816K), 0.2109783 secs] [Times: user=0.20 sys=0.00, real=0.20 secs]

5.122: [GC 5.122: [DefNew: 157247K->17472K(157248K), 0.2431866 secs] 291132K->289949K(506816K), 0.2432983 secs] [Times: user=0.25 sys=0.00, real=0.25 secs]

5.467: [GC 5.467: [DefNew: 157248K->157248K(157248K), 0.0000134 secs] 5.467: [Tenured: 272477K->349567K(349568K), 0.4970060 secs] 429725K->420767K(506816K), [Perm : 379K->379K(12288K)], 0.4971599 secs] [Times: user=0.48 sys=0.00, real=0.48 secs]

6.058: [Full GC 6.058: [Tenured: 349567K->41509K(349568K), 0.1328209 secs] 506815K->41509K(506816K), [Perm : 379K->379K(12288K)], 0.1329192 secs] [Times: user=0.14 sys=0.00, real=0.14 secs]

6.274: [GC 6.274: [DefNew: 139776K->17472K(157248K), 0.1368488 secs] 181285K->154007K(506816K), 0.1369451 secs] [Times: user=0.14 sys=0.00, real=0.14 secs]

6.495: [GC 6.495: [DefNew: 157248K->17471K(157248K), 0.1743854 secs] 293783K->268532K(506816K), 0.1744881 secs] [Times: user=0.19 sys=0.00, real=0.19 secs]

6.760: [GC 6.760: [DefNew: 157247K->157247K(157248K), 0.0000126 secs] 6.760: [Tenured: 251060K->349567K(349568K), 0.3610367 secs] 408308K->388050K(506816K), [Perm : 379K->379K(12288K)], 0.3611760 secs] [Times: user=0.37 sys=0.00, real=0.37 secs]

7.227: [Full GC 7.227: [Tenured: 349567K->33158K(349568K), 0.1144929 secs] 506815K->33158K(506816K), [Perm : 379K->379K(12288K)], 0.1145849 secs] [Times: user=0.13 sys=0.00, real=0.13 secs]

7.414: [GC 7.414: [DefNew: 139776K->17471K(157248K), 0.1112127 secs] 172934K->132730K(506816K), 0.1113000 secs] [Times: user=0.11 sys=0.00, real=0.11 secs]

7.599: [GC 7.599: [DefNew: 157247K->17472K(157248K), 0.1426927 secs] 272506K->234470K(506816K), 0.1427827 secs] [Times: user=0.14 sys=0.00, real=0.14 secs]

7.823: [GC 7.823: [DefNew: 157248K->157248K(157248K), 0.0000107 secs] 7.823: [Tenured: 216998K->340621K(349568K), 0.3163073 secs] 374246K->340621K(506816K), [Perm : 379K->379K(12288K)], 0.3164363 secs] [Times: user=0.31 sys=0.00, real=0.31 secs]

8.217: [GC 8.217: [DefNew: 139776K->139776K(157248K), 0.0000111 secs] 8.217: [Tenured: 340621K->349567K(349568K), 0.3631919 secs] 480397K->440326K(506816K), [Perm : 379K->379K(12288K)], 0.3633182 secs] [Times: user=0.36 sys=0.00, real=0.36 secs]

8.643: [Full GC 8.643: [Tenured: 349567K->37244K(349568K), 0.1115731 secs] 506815K->37244K(506816K), [Perm : 379K->379K(12288K)], 0.1116607 secs] [Times: user=0.11 sys=0.00, real=0.11 secs]

8.821: [GC 8.821: [DefNew: 139776K->17472K(157248K), 0.0952736 secs] 177020K->127601K(506816K), 0.0953525 secs] [Times: user=0.11 sys=0.00, real=0.11 secs]

8.983: [GC 8.983: [DefNew: 157248K->17471K(157248K), 0.1197464 secs] 267377K->219190K(506816K), 0.1198281 secs] [Times: user=0.11 sys=0.00, real=0.11 secs]

9.171: [GC 9.171: [DefNew: 157247K->17472K(157248K), 0.1685964 secs] 358966K->312990K(506816K), 0.1686990 secs] [Times: user=0.17 sys=0.00, real=0.17 secs]

9.467: [GC 9.467: [DefNew: 157248K->157248K(157248K), 0.0000150 secs] 9.467: [Tenured: 295518K->349567K(349568K), 0.5542238 secs] 452766K->452375K(506816K), [Perm : 379K->379K(12288K)], 0.5544066 secs] [Times: user=0.55 sys=0.00, real=0.55 secs]

10.113: [Full GC 10.114: [Tenured: 349567K->52002K(349568K), 0.1808479 secs]


```

506815K->52002K(506816K), [Perm : 379K->379K(12288K)], 0.1809750 secs] [Times:
user=0.19 sys=0.00, real=0.19 secs]
10.407: [GC 10.407: [DefNew: 139776K->17472K(157248K), 0.2176954 secs] 191778K->
190499K(506816K), 0.2178194 secs] [Times: user=0.22 sys=0.00, real=0.22 secs]
10.740: [GC 10.740: [DefNew: 157248K->17471K(157248K), 0.2684413 secs] 330275K->
329159K(506816K), 0.2685582 secs] [Times: user=0.28 sys=0.00, real=0.28 secs]
11.122: [GC 11.122: [DefNew: 157247K->157247K(157248K), 0.0000134 secs] 11.122:
[Tenured: 311687K->349567K(349568K), 0.5415732 secs] 468935K->462097K(506816K), [Perm :
379K->379K(12288K)], 0.5417346 secs] [Times: user=0.50 sys=0.00, real=0.55 secs]
11.757: [Full GC 11.757: [Tenured: 349567K->48817K(349568K), 0.1491027 secs]
506815K->48817K(506816K), [Perm : 379K->379K(12288K)], 0.1492014 secs] [Times:
user=0.16 sys=0.00, real=0.16 secs]
11.990: [GC 11.990: [DefNew: 139776K->17472K(157248K), 0.1398503 secs] 188593K->
162017K(506816K), 0.1399383 secs] [Times: user=0.14 sys=0.00, real=0.14 secs]
12.253: [GC 12.253: [DefNew: 157248K->17471K(157248K), 0.2044428 secs] 301793K->
293736K(506816K), 0.2045332 secs] [Times: user=0.20 sys=0.00, real=0.20 secs]
12.539: [GC 12.539: [DefNew: 157247K->157247K(157248K), 0.0000114 secs] 12.539:
[Tenured: 276264K->349567K(349568K), 0.3627016 secs] 433512K->403701K(506816K), [Perm :
379K->379K(12288K)], 0.3628280 secs] [Times: user=0.36 sys=0.00, real=0.36 secs]
12.986: [Full GC 12.986: [Tenured: 349567K->27685K(349568K), 0.1033825 secs]
506815K->27685K(506816K), [Perm : 379K->379K(12288K)], 0.1034725 secs] [Times:
user=0.11 sys=0.00, real=0.11 secs]
13.154: [GC 13.154: [DefNew: 139776K->17472K(157248K), 0.0961716 secs] 167461K->
116529K(506816K), 0.0962454 secs]

```

可以看到 1.14s 到 2.9s 直接出现了一次 Full GC，持续时间 0.17s，以及两个 minor gc，这些 gc 严重干扰了 PrintThread 的正常工作。

当通过 Stop-the-world 机制的方式来运行垃圾收集器时，垃圾收集器会在内存回收的过程中暂停程序中所有的工作线程，直至完成内存回收才会恢复之前被暂停的工作线程。如果 Stop-the-world 出现在新生代的 Minor GC 中时，由于新生代的内存空间通常都比较小，所以暂停时间也在可接受的合理范围之内，不过一旦出现在老年代的 Full GC 中时，程序的工作线程被暂停的时间将会更久，这往往直接跟 Java 堆空间所管理的内存大小有关。简单来说，内存空间越大，执行 Full GC 的时间就会越久，相对的工作线程被暂停的时间也就会更长。以互联网项目为例，由于项目的特殊性，因此经常需要用到多大几十乃至上百 GB 的内存，如果用户的登录操作恰巧碰见垃圾收集器在执行 Full GC 时，这将会是一场灾难，如果用户等待的时间过长，这就不是用户体验下降这么简单的事情了，甚至有可能会流失用户，所以 JVM 的设计者们提供了并发回收希望以此缩短 Stop-the-world 机制的暂停时运行或交叉运行。直到目前为止，哪怕是 G1 也不能完全避免 Stop-the-world 情况发生，只能说垃圾回收器越来越优秀，回收效率越来越高，尽可能地缩短了暂停时间。

7.3.2 垃圾回收算法

目前有两种比较常见的垃圾标记算法，分别是引用计数算法和根搜索算法。引用计数器在微软的 COM 组件技术中、Adodb 的 ActionScript3 种都有使用。

7.3.2.1 引用计数法

引用计数法（Reference Counting）在 GC 执行垃圾回收之前，首先需要区分出内存中哪些是存活对象，哪些是已经死亡的对象，只有被标记为已经死亡的对象，GC 才会在执行垃圾回收时，释放掉其所占用的内存空间，因此这个过程我们可以称之为垃圾标记阶段。

引用计数器的实现很简单，对于一个对象 A，只要有任何一个对象引用了 A，则 A 的引用计数器就加 1，当引用失效时，引用计数器就减 1。只要对象 A 的引用计数器的值为 0，则对象 A 就不可能再被使用。也就是说，引用计数器的实现只需要为每个对象配置一个整形的计数器即可。引用计数器算法的一大优势就是不用等待内存不够用的时候，才进行垃圾的回收，完全可以在赋值操作的同时检查计数器是否为 0，如果是的话就可以立即回收。

但是引用计数器有一个严重的问题，即无法处理循环引用的情况。一个简单的循环引用问题描述如下：有对象 A 和对象 B，对象 A 中含有对象 B 的引用，对象 B 中含有对象 A 的引用。此时，对象 A 和对象 B 的引用计数器都不为 0。但是在系统中却不存在任何第 3 个对象引用了 A 或 B。也就是说，A 和 B 是应该被回收的垃圾对象，但由于垃圾对象间相互引用，从而使垃圾回收器无法识别，引起内存泄漏。

如图 7-2 所示，我们构造了一个列表，我们将最后一个元素的 next 属性指向第一个元素，即引用第一个元素，从而构成循环引用；这个时候如果我们将列表的头 head 赋值为 null，此时列表的各个元素的计数器都不为 0，同时我们也失去了对列表的引用控制，从而导致列表元素不能被回收！

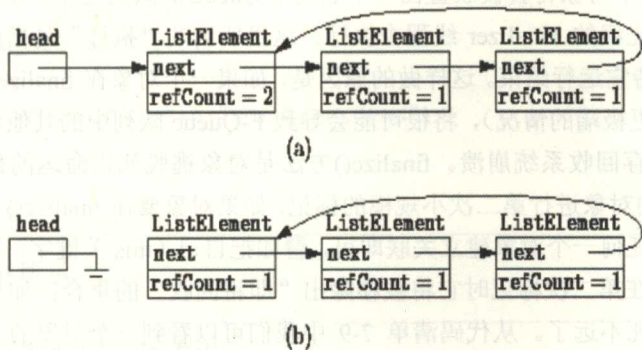


图7-2 引用计数流程图

引用计数器拥有一些特性，首先它需要单独的字段存储计数器，这样的做法增加了存储空间开销。其次每次赋值都需要更新计数器，这增加了时间开销。再者垃圾对象便于辨识，只要计数器为 0，就可作为垃圾回收。接下来它方便及时回收垃圾，没有延迟性。最后不能解决循环引用的问题。正是由于最后一条致命缺陷，导致在 Java 的垃圾回收器中没有使用这类算法。

7.3.2.2 根搜索算法

HotSpot 和大部分 JVM 中是使用根搜索算法作为垃圾标记的算法实现。前面介绍过的引用计数算法尽管实现简单，执行效率也不错，但是该算法本身却存在一个较大的弊端，甚至会影响到垃圾标记的准确性。由于引用计数算法会为程序中的每一个对象都创建一个私有的引用计数器，当目标对象被其他存活对象引用时，引用计数器中的值则会加 1，不再引用时便会减 1，当引用计数器中的值为 0 的时候，就意味着该对象已经不再被任何存活对象引用，可以被标记为垃圾对象。

采用这种方式看起来似乎没有任何问题，但是如果一些明显已经死亡了的对象尽管没有被任何的存活对象引用，但是它们彼此之间却存在相互引用时，引用计数器中的值则永远不会为 0，这样便会导致 GC 在执行内存回收时永远无法释放掉无用对象所占用的内存空间，极有可能引发内存泄露。

相对于引用计数算法而言，根搜索算法不仅同样具备实现简单和执行高效等特点，更重要的是该算法可以有效地解决在引用计数算法中一些已经死亡的对象因相互引用而导致的无法正确被标记的问题，防止内存泄露的发生。简单来说，根搜索算法是以根对象集合作为起始点，按照从上至下的方式搜索被根对象集合所连接的目标对象是否可达（使用根搜索算法后，内存中的存活对象都会被根对象集合直接或间接连接着），如果目标对象不可达时，就意味着该对象已经死亡，便可以在 `instanceOopDesc`⁴ 的 `Mark World` 中将其标记为垃圾对象。在根搜索算法中，只有能够被根对象集合直接或者间接连接的对象才是存活对象。在 `HotSpot` 中，根对象集合中包含了 5 个元素，Java 栈中的对象引用、本地方法栈中的对象引用、运行时常量池中的对象引用、方法区中类静态属性的对象引用以及和一个类对应的唯一数据类型的 `Class` 对象。

注意，在根搜索算法中不可达的对象，也并非是非“非死不可”的，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程：如果对象在进行根搜索后发现没有与 GC Roots 相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 `finalize()` 方法。当对象没有覆盖 `finalize()` 方法，或者 `finalize()` 方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。如果这个对象被判定为有必要执行 `finalize()` 方法，那么这个对象将会被放置在一个名为 `F-Queue` 的队列之中，并在稍后由一条由虚拟机自动建立的、低优先级的 `Finalizer` 线程去执行。这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束。这样做的原因是，如果一个对象在 `finalize()` 方法中执行缓慢，或者发生了死循环（更极端的情况），将很可能会导致 `F-Queue` 队列中的其他对象永久处于等待状态，甚至导致整个内存回收系统崩溃。`finalize()` 方法是对对象逃脱死亡命运的最后一次机会，稍后 GC 将对 `F-Queue` 中的对象进行第二次小规模标记，如果对象要在 `finalize()` 中成功拯救自己一只要重新与引用链上的任何一个对象建立关联即可，譬如把自己（`this` 关键字）赋值给某个类变量或对象的成员变量，那在第二次标记时它将被移除出“即将回收”的集合；如果对象这时候还没有逃脱，那它就真的离死不远了。从代码清单 7-9 中我们可以看到一个对象的 `finalize()` 被执行，但是它仍然可以存活。

代码清单 7-9 逃脱回收实验

```
public class FinalizeEscapeGC {
    public static FinalizeEscapeGC SAVE_HOOK = null;
    public void isAlive() {
        System.out.println("yes, i am still alive");
    }

    @Override
```

⁴ `HotSpot` 在 C++ 代码中用 `instanceOopDesc` 类来表示 Java 对象，而该类继承 `oopDesc`，所以 `HotSpot` 中的 Java 对象也自然拥有 `oopDesc` 所声明的头部。


```

protected void finalize() throws Throwable {
    super.finalize();
    System.out.println("finalize mehtod executed!");
    FinalizeEscapeGC.SAVE_HOOK = this;
}

public static void main(String[] args) throws Throwable {
    SAVE_HOOK = new FinalizeEscapeGC();
    //对象第一次成功拯救自己
    SAVE_HOOK = null;
    System.gc();
    // 因为 Finalizer 方法优先级很低, 暂停 0.5 秒, 以等待它
    Thread.sleep(500);
    if (SAVE_HOOK != null) {
        SAVE_HOOK.isAlive();
    } else {
        System.out.println("no, i am dead");
    }
    // 下面这段代码与上面的完全相同, 但是这次自救却失败了
    SAVE_HOOK = null;
    System.gc();
    // 因为 Finalizer 方法优先级很低, 暂停 0.5 秒, 以等待它
    Thread.sleep(500);
    if (SAVE_HOOK != null) {
        SAVE_HOOK.isAlive();
    } else {
        System.out.println("no, i am dead");
    }
}
}

```

输出如清单 7-10 所示。

代码清单 7-10 逃脱回收实验运行输出

```

finalize mehtod executed!
yes, i am still alive
no, i am dead

```

从代码清单 7-10 的运行结果可以看到, SAVE_HOOK 对象的 finalize() 方法确实被 GC 收集器触发过, 并且在被收集前成功逃脱了。另外一个值得注意的地方就是, 代码中有两段完全一样的代码片段, 执行结果却是一次逃脱成功, 一次失败, 这是因为任何一个对象的 finalize() 方法都只会被系统自动调用一次, 如果对象面临下一次回收, 它的 finalize() 方法不会被再次执行, 因此第二段代码的自救行动失败了。

7.3.2.3 标记——清除算法 (Mark-Sweep)

当成功区分出内存中存活对象和死亡对象后, GC 接下来的任务就是执行垃圾回收释放掉无用对象所占用的内存空间, 以便有足够的可用内存空间为新对象分配内存。目前在 JVM 中比较常见

的三种垃圾收集算法是标记--清除算法（Mark-Sweep）、复制算法（Copying）、标记一压缩算法（Mark-Compact）。在介绍三种算法之前，我们先来通过表 7-1 看看它们之间的区别。

表 7-1 算法比较表

	mark-sweep	mark-compact	copying
速度	中等	最慢	最快
空间开销	少（但会堆积碎片）	少（不堆积碎片）	通常需要活对象的 2 倍大小（不堆积碎片）
移动对象	否	是	是

标记--清除算法（Mark-Sweep）是一种非常基础和常见的垃圾收集算法，该算法被 J.McCarthy 等人在 1960 年提出并成功地发明并应用于 Lisp 语言。我们以餐巾纸作为示例，午餐过程中，餐厅里的所有人都根据自己的需要取用餐巾纸。当垃圾收集机器人想收集废旧餐巾纸的时候，它会让所有用餐的人先停下来，然后，依次询问餐厅里的每一个人：“你正在用餐巾纸吗？你用的是哪一张餐巾纸？”机器人根据每个人的回答将人们正在使用的餐巾纸画上記号。询问过程结束后，机器人在餐厅里寻找所有散落在餐桌上且没有记号的餐巾纸（这些显然都是用过的废旧餐巾纸），把它们统统扔到垃圾箱里。

回到算法本身。算法涉及几个概念，我们先来了解一下 mutator 和 collector，这两个名词经常在垃圾收集算法中出现，collector 指的就是垃圾收集器，而 mutator 是指除了垃圾收集器之外的部分，比如说我们应用程序本身。mutator 的职责一般是 NEW(分配内存),READ(从内存中读取内容),WRITE(将内容写入内存)，而 collector 则就是回收不再使用的内存来供 mutator 进行 NEW 操作的使用。mutator 根对象一般指的是分配在堆内存之外，可以直接被 mutator 直接访问到的对象，一般是指静态/全局变量以及 Thread-Local 变量⁵。

标记-清除算法将垃圾回收分为两个阶段，标记阶段和清除阶段。在标记阶段，collector 从 mutator 根对象开始进行遍历，对从 mutator 根对象可以访问到的对象都打上一个标识，一般是在对象的 header 中，将其记录为可达对象。而在清除阶段，collector 对堆内存(heap memory)从头到尾进行线性的遍历，如果发现某个对象没有标记为可达对象，通过读取对象的 header 信息，则就将其回收。一种可行的实现是，在标记阶段首先通过根节点，标记所有从根节点开始的各大对象。因此，未被标记的对象就是未被引用的垃圾对象。然后，在清除阶段，清除所有未被标记的对象。

前面说过，标记一清除算法的执行过程分为“标记”和“清除”两大阶段。这种分步执行的思路奠定了现代垃圾收集算法的思想基础。与引用计数算法不同的是，标记一清除算法不需要运行环境监测每一次内存分配和指针操作，而只要在“标记”阶段中跟踪每一个指针变量的指向，用类似思路实现的垃圾收集器也常被后人统称为跟踪收集器（Tracing Collector）。

标记--清除算法最大的问题是存在大量的空间碎片，因为回收后的空间是不连续的。在对象的堆空间分配过程中，尤其是大对象的内存分配，不连续的内存空间的工作效率要低于连续的空间。

相对于另外两种内存回收算法而言，标记--清除算法（Mark-Sweep）不仅执行效率低下，更重要的是，由于被执行内存回收的无用对象所占用的内存空间有可能是一些不连续的内存块，不

⁵ 在 Java 中，存储在 java.lang.ThreadLocal 中的变量和分配在栈上的变量方法内部的临时变量等都属于此类。

可避免地会产生一些内存碎片，从而导致后续没有足够的可用内存空间分配给较大的对象。

7.3.2.4 复制算法 (Copying)

为了解决标记—清除算法在垃圾收集效率方面的缺陷，M.L.Minsky 于 1963 年发表了著名的论文“一种使用双存储区的 Lisp 语言垃圾收集器 (A LISP Garbage Collector Algorithm Using Serial Secondary Storage)”。M.L.Minsky 在该论文中描述的算法被人们称为复制算法，它也被 M.L.Minsky 本人成功地引入到了 Lisp 语言的一个实现版本中。标记—清除算法后来被引入 JVM 中，提升了 GC 在垃圾标记和内存释放这两个阶段的执行效率。还是采用之前的餐厅示例，餐厅被垃圾收集机器人分成南区 and 北区两个大小完全相同的部分。午餐时，所有人都先在南区用餐（因为空间有限，用餐人数自然也将减少一半），用餐时可以随意使用餐巾纸。当垃圾收集机器人认为有必要回收废旧餐巾纸时，它会要求所有用餐者以最快的速度从南区转移到北区，同时随身携带自己正在使用的餐巾纸。等所有人都转移到北区之后，垃圾收集机器人只要简单地把南区中所有散落的餐巾纸扔进垃圾箱就算完成任务了。下一次垃圾收集的工作过程也大致类似，唯一的不同只是人们的转移方向变成了从北区到南区。如此循环往复，每次垃圾收集都只需简单地转移（也就是复制）一次，垃圾收集速度无与伦比——当然，对于用餐者往返奔波于南北两区之间的辛劳，垃圾收集机器人是决不会流露出丝毫怜悯的。

回到算法本身。复制算法首先将活着的内存空间分为两块，每次只使用其中一块，在垃圾回收时将正在使用的内存中的存活对象复制到未被使用的内存块中，之后清除正在使用的内存块中的所有对象，交换两个内存的角色，最后完成垃圾回收。

如果系统中的垃圾对象很多，复制算法需要复制的存活对象数量并不会太大。因此在真正需要垃圾回收的时刻，复制算法的效率是很高的。又由于对象在垃圾回收过程中统一被复制到新的内存空间中，因此，可确保回收后的内存空间是没有碎片的。该算法的缺点是将系统内存折半。

Java 的年轻代串行垃圾回收器中使用了复制算法的思想。年轻代分为 eden 空间、from 空间、to 空间 3 个部分。其中 from 空间和 to 空间可以视为用于复制的两块大小相同、地位相等，且可进行角色互换的空间块。from 和 to 空间也称为 survivor 空间，即幸存者空间，用于存放未被回收的对象。

在垃圾回收时，eden 空间中的存活对象会被复制到未使用的 survivor 空间中(假设是 to)，正在使用的 survivor 空间(假设是 from)中的年轻对象也会被复制到 to 空间中(大对象，或者老年对象会直接进入老年带，如果 to 空间已满，则对象也会直接进入年老代)。此时，eden 空间和 from 空间中的剩余对象就是垃圾对象，可以直接清空，to 空间则存放此次回收后的存活对象。这种改进的复制算法既保证了空间的连续性，又避免了大量的内存空间浪费。

基于分代的概念，Java 堆区如果还要更进一步细分的话，还可以划分为新生代 (YoungGen) 和老年代 (OldGen)，其中新生代又可以被划分为 Eden 空间、From Survivor 空间和 To Survivor 空间。在 HotSpot 中，Eden 空间和另外两个 Survivor 空间默认所占的比例是 8: 1，当然开发人员可以通过选项 “-XX:SurvivorRatio” 调整这个空间比例。当执行一次 Minor GC（新生代的垃圾回收）时，Eden 空间中的存活对象会被复制到 To 空间内，并且之前已经经历过一次 Minor GC 并在 From 空间中存活下来的对象如果还年轻的话同样也会被复制到 To 空间内。需要注意的是，在满足两种特殊情况下，Eden 和 From 空间中的存活对象将不会被复制到 To 空间内。首先是如果存活

对象的分代年龄超过选项“-XX: MaxTenuringThreshold”所指定的阈值时，将会直接晋升到老年代中；其次当 To 空间的容量达到阈值时，存活对象同样也是直接晋升到老年代中。当所有的存活对象都被复制到 To 空间或者晋升到老年代后，剩下的均为垃圾对象，这就意味着 GC 可以对这些已经死亡了的对象执行一次 Minor GC，释放掉其所占用的内存空间。

当执行完 Minor GC 之后，Eden 空间和 From 空间将会被清空，而存活下来的对象则会被全部存储在 To 空间内，接下来 From 空间和 To 空间将会互换位置。其实复制算法无非就是使用 To Survivor 空间作为一个临时的空间交换角色，务必需要保证两块 Survivor 空间中一块必须是空的，这就是复制算法。尽管复制算法能够高效执行 Minor GC，但是它却并不适用于老年代中的内存回收，因为老年代中对象的生命周期都比较长，甚至在某些极端的情况下还能够与 JVM 的生命周期保持一致，所以如果老年代也采用复制算法执行内存回收不仅需要额外的时间和空间，而且还会导致较多的复制操作影响到 GC 的执行效率。

总的来说，由于 JVM 中的绝大多数对象都是瞬时状态的，生命周期非常短暂，所以复制算法被广泛应用于新生代中。分区、复制的思路不仅大幅提高了垃圾收集的效率，而且也将原本繁纷复杂的内存分配算法变得前所未有的简明扼要（既然每次内存回收都是对整个半区的回收，内存分配时也就不考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存就可以了），这简直是个奇迹！不过，任何奇迹的出现都有一定的代价，在垃圾收集技术中，复制算法提高效率的代价是人为地将可用内存缩小了一半。

7.3.2.5 标记—压缩算法 (Mark-Compact)

标记—清除算法的确可以应用在老年代中，但是该算法不仅执行效率低下，而且在执行完内存回收后还会产生内存碎片，所以 JVM 的设计者们在此基础上进行了改进，标记—压缩算法由此诞生。标记—整理算法是标记—清除算法和复制算法的有机结合。把标记—清除算法在内存占用上的优点和复制算法在执行效率上的特长综合起来，这是所有人都希望看到的结果。不过，两种垃圾收集算法的整合并不像 1 加 1 等于 2 那样简单，我们必须引入一些全新的思路。1970 年前后，G. L. Steele, C. J. Chene 和 D. S. Wise 等研究者陆续找到了正确的方向，标记—整理算法的轮廓也逐渐清晰了起来。还是采用之前的餐厅示例，在我们熟悉的餐厅里，这一次，垃圾收集机器人不再把餐厅分成两个南北区域了。需要执行垃圾收集任务时，机器人先执行标记—清除算法的第一个步骤，为所有使用中的餐巾纸画好标记，然后，机器人命令所有就餐者带上标记的餐巾纸向餐厅的南面集中，同时把没有标记的废旧餐巾纸扔向餐厅北面。这样一来，机器人只要站在餐厅北面，怀抱垃圾箱，迎接扑面而来的废旧餐巾纸就行了。

回到算法本身。当成功标记出内存中的垃圾对象后，标记—压缩算法会将所有的存活对象移动到一个规整且连续的内存空间中，然后执行 Full GC（老年代的垃圾回收，或者被称为 Major GC）回收无用对象所占用的内存空间。当成功执行压缩之后，已用和未用的内存都各占一边，彼此之间维系着一个记录下一次分配起始点的标记指针，当为新对象分配内存时，则可以使用指针碰撞（Bump the Pointer）技术修改指针的偏移量将新对象分配在第一个空闲内存位置上，为新对象分配内存带来便捷。

在 HotSpot 中，基于分代的概念，GC 所使用的内存回收算法必须结合新生代和老年代各自的特点，简单来说，就是针对不同的代空间，从而结合使用不同的垃圾收集算法。为新生代选择的垃圾收集算法通常是以速度优先，因为新生代中所存储的瞬时对象声明周期非常短暂，可以有针

对性地使用复制算法,因此执行 Minor GC 时,一定要保持高效和快速。而新生代中的生存空间通常都比较小,所以回收新生代时一定会非常频繁。但老年代通常使用更节省内存的回收算法,因为老年代中所存储的对象声明周期都非常长,并且老年代占据了大部分的堆空间,所以老年代的 Full GC 并不会跟新生代的 Minor GC 一样频繁,不过一旦程序中发生一次 Full GC 时,将会耗费更长的时间来完成,那么在老年代中使用标记-清除算法或者标记-压缩算法执行垃圾回收将会是不错的选择。

复制算法的高效性是建立在存活对象少、垃圾对象多的前提下的。这种情况在年轻代经常发生,但是在年老代更常见的情况是大部分对象都是存活对象。如果依然使用复制算法,由于存活的对象较多,复制的成本也很高。标记-压缩算法是一种年老代的回收算法,它在标记-清除算法的基础上做了一些优化。也首先需要从根节点开始对所有可达对象做一次标记,但之后,它并不简单地清理未标记的对象,而是将所有的存活对象压缩到内存的一端。之后,清理边界外所有的空间。这种方法既避免了碎片的产生,又不需要两块相同的内存空间,因此,其性价比较高。

标记-整理算法的总体执行效率高于标记-清除算法,又不像复制算法那样需要牺牲一半的存储空间,这显然是一种非常理想的结果。在许多现代的垃圾收集器中,人们都使用了标记-整理算法或其改进版本。

7.3.2.6 增量算法 (Incremental Collecting)

在垃圾回收过程中,应用软件将处于一种 Stop the World 的状态。在 Stop the World 状态下,应用程序所有的线程都会挂起,暂停一切正常的工作,等待垃圾回收的完成。如果垃圾回收时间过长,应用程序会被挂起很久,将严重影响用户体验或者系统的稳定性。为了解决这个问题,即对实时垃圾收集算法的研究直接导致了增量收集算法的诞生。

最初,为了进行实时的垃圾收集,可以设计一个多进程的运行环境,比如用一个进程执行垃圾收集工作,另一个进程执行程序代码。这样一来,垃圾收集工作看上去就仿佛是在后台悄悄完成的,不会打断程序代码的运行。在收集餐巾纸的例子中,这一思路可以被理解为,垃圾收集机器人在人们用餐的同时寻找废弃的餐巾纸并将它们扔到垃圾箱里。这个看似简单的思路会在设计和实现时碰上进程间冲突的难题。比如说,如果垃圾收集进程包括标记和清除两个工作阶段,那么,垃圾收集器在第一阶段中辛辛苦苦标记出的结果很可能被另一个进程中的内存操作代码修改得面目全非,以至于第二阶段的工作没有办法开展。

M. L. Minsky 和 D. E. Knuth 对实时垃圾收集过程中的技术难点进行了早期的研究,G. L. Steele 于 1975 年发表了题为“多进程整理的垃圾收集(Multiprocessing Compactifying Garbage Collection)”的论文,描述了一种被后人称为“Minsky-Knuth-Steele 算法”的实时垃圾收集算法。E.W.Dijkstra, L.Lamport, R.R.Fenichel 和 J.C.Yochelson 等人也相继在此领域做出了各自的贡献。1978 年, H.G.Baker 发表了“串行计算机上的实时表处理技术(List Processing in Real Time on a Serial Computer)”一文,系统阐述了多进程环境下用于垃圾收集的增量收集算法。

增量算法的基本思想是,如果一次性将所有的垃圾进行处理,需要造成系统长时间的停顿,那么就可以让垃圾收集线程和应用程序线程交替执行。每次,垃圾收集线程只收集一小片区域的内存空间,接着切换到应用程序线程。依次反复,直到垃圾收集完成。使用这种方式,由于在垃圾回收过程中,间断性地执行了应用程序代码,所以能减少系统的停顿时间。但是,因为线程切

换和上下文转换的消耗，会使得垃圾回收的总体成本上升，造成系统吞吐量的下降。

总的来说，增量收集算法的基础仍是传统的标记—清除和复制算法。增量收集算法通过对进程间冲突的妥善处理，允许垃圾收集进程以分阶段的方式完成标记、清理或复制工作。

7.3.2.7 分代收集算法 (Generational Collecting)

1980 年前后，善于在研究中使用统计分析知识的技术人员发现，大多数内存块的生存周期都比较短，垃圾收集器应当把更多的精力放在检查和清理新分配的内存块上。还是餐巾纸的例子，如果垃圾收集机器人足够聪明，事先摸清了餐厅里每个人在用餐时使用餐巾纸的习惯——比如有些人喜欢在用餐前后各用掉一张餐巾纸，有的人喜欢自始至终攥着一张餐巾纸不放，有的人则每打一个喷嚏就用去一张餐巾纸——机器人就可以制定出更完善的餐巾纸回收计划，并总是在人们刚扔掉餐巾纸没多久就把垃圾捡走。这种基于统计学原理的做法当然可以让餐厅的整洁度成倍提高。D. E. Knuth, T. Knight, G. Sussman 和 R. Stallman 等人对内存垃圾的分类处理做了最早的研究。1983 年，H. Lieberman 和 C. Hewitt 发表了题为“基于对象寿命的一种实时垃圾收集器 (A real-time garbage collector based on the lifetimes of objects)”的论文。这篇著名的论文标志着分代收集算法的正式诞生。此后，在 H. G. Baker, R. L. Hudson, J. E. B. Moss 等人的共同努力下，分代收集算法逐渐成为了垃圾收集领域里的主流技术。

根据垃圾回收对象的特性，使用合适的算法回收，分代既是基于这种思想，它将内存区间根据对象的特点分成几块，根据每块内存区间的特点，使用不同的回收算法，以提高垃圾回收的效率。

以 Hot Spot 虚拟机为例，它将所有的新建对象都放入称为年轻代的内存区域，年轻代的特点是对象会很快回收，因此，在年轻代就选择效率较高的复制算法。当一个对象经过几次回收后依然存活，对象就会被放入称为年老代的内存空间。在年老代中，几乎所有的对象都是经过几次垃圾回收后依然得以幸存的。因此，可以认为这些对象在一段时期内，甚至在应用程序的整个生命周期中，将是常驻内存的。如果依然使用复制算法回收年老代，将需要复制大量对象。再加上年老代的回收性价比也要低于年轻代，因此这种做法也是不可取的。根据分代的思想，可以对年老代的回收使用与年轻代不同的标记-压缩算法，以提高垃圾回收效率。

总的来说，分代收集算法是基于对对象生命周期分析后得出的垃圾回收算法。它把对象分为年青代、年老代、持久代，对不同生命周期的对象使用不同的算法（上述方式中的一个）进行回收。JVM 垃圾回收器（从 J2SE1.2 开始）都是使用此算法的。

7.3.3 垃圾收集器

7.3.3.1 垃圾收集器分类

由于 JDK 的版本处于高速迭代过程中，因此 Java 从发展至今已经衍生了众多的 GC 版本，比如 Serial/Serial Old 收集器、ParNew 收集器、Parallel/Parallel Old 收集器、CMS (Concurrent-Mark-Sweep) 收集器，以及从 JDK7 Update4 版本开始提供的 G1 (Garbage-First) 收集器等。

基于分代的概念，不同的代空间中均活动着不同的 GC，比如 Serial 收集器就是一个典型的新生代垃圾收集器，它采用复制算法回收新生代无用对象的内存空间。当然，JVM 在实际运行过程中，新生代和老年代中各自的 GC 需要组合在一起共同执行垃圾回收任务。如果新生代的 GC 和

老年代的 GC 相连，则意味着可以组合在一起使用，不过在实际开发过程中，新生代和老年代的 GC 的组合方式还需要结合具体的应用场景进行分析后得到。

从不同角度分析垃圾收集器，可以将 GC 分为不同的类型。

按线程数分，可以分为串行垃圾回收器和并行垃圾回收器。串行垃圾回收器一次只使用一个线程进行垃圾回收；并行垃圾回收器一次将开启多个线程同时进行垃圾回收。在并行能力较强的 CPU 上，使用并行垃圾回收器可以缩短 GC 的停顿时间。

按照工作模式分，可以分为并发式垃圾回收器和独占式垃圾回收器。并发式垃圾回收器与应用程序线程交替工作，以尽可能减少应用程序的停顿时间；独占式垃圾回收器(Stop the world)一旦运行，就停止应用程序中的其他所有线程，直到垃圾回收过程完全结束。

按碎片处理方式可分为压缩式垃圾回收器和非压缩式垃圾回收器。压缩式垃圾回收器会在回收完成后，对存活对象进行压缩整理，消除回收后的碎片；非压缩式的垃圾回收器不进行这步操作。

按工作的内存区间，又可分为年轻代垃圾回收器和年老代垃圾回收器。下面开始对这些型式进行逐一讨论。

7.3.3.2 串行回收器和并行回收器

串行回收指的是在同一时间段内只允许一件事情发生，简单来说，当多个 CPU 可用时，也只能有一个 CPU 用于执行垃圾回收操作，并且在执行垃圾回收时，程序中的工作线程将会被暂停，当垃圾收集工作完成后才会恢复之前被暂停的工作线程，这就是串行回收。不过由于运行在客户端下的 Client 模式的 JVM 并没有低暂停时间的要求，并且 Client 模式下的内存开销相对于 Server 模式来说也更小，因此串行回收默认被应用在 Client 模式下的 JVM 中。和串行回收相反，并行收集可以运用多个 CPU 同时执行垃圾回收，因此提升了应用的吞吐量，不过并行回收仍然使用了“Stop-the-world”机制和复制算法。

串行收集器主要有两个特点，首先，它仅仅使用单线程进行垃圾回收。其次，它独占式的垃圾回收方式。

在串行收集器进行垃圾回收时，Java 应用程序中的线程都需要暂停，等待垃圾回收的完成，这样给用户体验造成较差效果。虽然如此，串行收集器却是一个成熟、经过长时间生产环境考验的极为高效的收集器。年轻代串行处理器使用复制算法，实现相对简单，逻辑处理特别高效，且没有线程切换的开销。在诸如单 CPU 处理器或者较小的应用内存等硬件平台不是特别优越的场合，它的性能表现可以超过并行回收器和并发回收器。

并行收集器是工作在年轻代的垃圾收集器，它只简单地将串行回收器多线程化。它的回收策略、算法以及参数和串行回收器一样。

并行回收器也是独占式的回收器，在收集过程中，应用程序会全部暂停。但由于并行回收器使用多线程进行垃圾回收，因此，在并发能力比较强的 CPU 上，它产生的停顿时间要短于串行回收器，而在单 CPU 或者并发能力较弱的系统中，并行回收器的效果不会比串行回收器好，由于多线程的压力，它的实际表现很可能比串行回收器差。

7.3.3.3 Serial 收集器

Serial 收集器作用于新生代中，它采用复制算法、串行回收和“Stop-the-World”机制的方式执行内存回收。在早期的 JDK 版本中，由于那个年代的 CPU 速度并没有这么快，所以在 CPU 受限于单个 CPU 的情况下，使用 Serial 收集器执行新生代垃圾收集几乎是唯一的选择，并且 Serial 收集器默认也作为 HotSpot 中 Client 模式下的新生代垃圾收集器。

除了新生代之外，Serial 收集器还提供用于执行老年代垃圾收集的 Serial Old 收集器。Serial Old 收集器同样也采用了串行回收和“Stop-the-World”机制，只不过内存回收算法使用电俄式标记-压缩算法。在此需要注意的是，如果 JVM 受限于单个 CPU 的环境下，使用 Serial 收集器加上 Serial Old 收集器的组合执行 Client 模式下的内存回收将会是不错的选择。基于串行回收的垃圾收集器适用于大多数对暂停时间要求不高的 Client 模式下的 JVM，由于 CPU 不需要频繁地做任务切换，因此可以有效避免多线程交互过程中产生的一些额外开销，虽然执行串行回收会降低程序的吞吐量，但是回收质量还是不错的。在程序中，开发人员可以通过选项“-XX:+UseSerialGC”手动指定使用 Serial 收集器执行内存回收任务。

我们可以把上面的内容总结如下：

- (1) 该算法的第一步是在老年代标记存活的对象；
- (2) 从头开始检查堆内存空间，并且只留下依然幸存的对象（清除）；
- (3) 最后一步，从头开始，顺序地填满堆内存空间，将存活的对象连续存放在一起，这样堆分成两部分，一边有存放的对象，一边没有对象（整理）；
- (4) Serial 收集器应用于小的存储器和少量的 CPU。

■ 年轻代串行收集器

在 HotSpot 虚拟机中，使用-XX: +UseSerialGC 参数可以指定使用年轻代串行收集器和年老代串行收集器。当 JVM 在 Client 模式下运行时，它是默认的垃圾收集器。

清单 7-11 所示是一次年轻代串行收集器的工作输出日志，信息如下所示（使用-XX:+PrintGCDetails 开关）。

代码清单 7-11 年轻代串行收集器工作输出

```
[GC [DefNew: 3468K->150K(9216K), 0.0028638 secs][Tenured: 1562K->1712K(10240K),  
0.0084220 secs] 3468K->1712K(19456K), [Perm : 377K->377K(12288K)], 0.0113816 secs]  
[Times: user=0.02 sys=0.00, real=0.01 secs]
```

上面的输出显示了一次垃圾回收前的年轻代的内存占用量和垃圾回收后的年轻代内存占用量，以及垃圾回收所消耗的时间。

■ 年老代串行收集器

前面提起过，年老代串行收集器使用的是标记-压缩算法。和年轻代串行收集器一样，它也是一个串行的、独占式的垃圾回收器。由于年老代垃圾回收通常会使用比年轻代垃圾回收更长的时间，因此，在堆空间较大的应用程序中，一旦年老代串行收集器启动，应用程序很可能会因此停顿几秒甚至更长时间。

虽然如此，年老代串行回收器可以和多种年轻代回收器配合使用，同时它也可以作为 CMS 回收器的备用回收器。

若要启用年老代串行回收器，可以尝试使用以下参数：

-XX:+UseSerialGC：年轻代、年老代都使用串行回收器；

输出如清单 7-12 所示。

代码清单 7-12 使用-XX:+UseSerialGC 参数

```
Heap
def new generation    total 4928K, used 1373K [0x27010000, 0x27560000, 0x2c560000)
  eden space 4416K,    31% used [0x27010000, 0x27167628, 0x27460000)
  from space 512K,     0% used [0x27460000, 0x27460000, 0x274e0000)
  to   space 512K,     0% used [0x274e0000, 0x274e0000, 0x27560000)
tenured generation    total 10944K, used 0K [0x2c560000, 0x2d010000, 0x37010000)
  the space 10944K,    0% used [0x2c560000, 0x2c560000, 0x2c560200, 0x2d010000)
compacting perm gen   total 12288K, used 376K [0x37010000, 0x37c10000, 0x3b010000)
  the space 12288K,    3% used [0x37010000, 0x3706e0b8, 0x3706e200, 0x37c10000)
  ro space 10240K,     51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
  rw space 12288K,     55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

我们也可以使用-XX:+UseParNewGC 参数，当使用它作为 JVM 参数时，指定年轻代使用并行收集器，年老代使用串行收集器。输出如下所示。

7.3.3.4 ParNew 收集器

如果说 Serial 是新生代中的单线程垃圾收集器，那么 ParNew 收集器则是 Serial 收集器的多线程版本。ParNew 收集器除了采用并行回收的方式执行内存回收外，两款垃圾收集器之间几乎没有任何区别，因为 ParNew 收集器在新生代中同样也是采用复制算法和“Stop-the-World”机制。

如果说 ParNew 收集器运行在多 CPU 的环境下，由于可以充分利用多 CPU、多核心等物理硬件资源优势，确实可以更快速地完成垃圾收集，提升程序的吞吐量，但是如果是在单个 CPU 的环境下，ParNew 收集器不见得比 Serial 收集器更高效。虽然 Serial 收集器是基于串行回收，但是由于 CPU 不需要频繁地做任务切换，因此可以有效避免多线程交互过程中产生的一些额外开销。所以从理论上来说，Serial 收集器的优势是在 JVM 受限于单 CPU 环境中，而 ParNew 收集器的优势则是体现在多 CPU、多核心的环境中，并且在某些注重低延迟的应用场景下，ParNew 收集器和 CMS（Concurrent-Mark-Sweep）收集器的组合模式，在 Server 模式下的内存回收效果很好。在程序中，开发人员可以通过选项“-XX:+UseParNewGC”手动指定使用 ParNew 收集器执行内存回收任务。

代码清单 7-13 使用-XX:+UseParNewGC 参数

```
Heap
par new generation    total 4928K, used 1373K [0x0f010000, 0x0f560000, 0x14560000)
  eden space 4416K,    31% used [0x0f010000, 0x0f167620, 0x0f460000)
  from space 512K,     0% used [0x0f460000, 0x0f460000, 0x0f4e0000)
  to   space 512K,     0% used [0x0f4e0000, 0x0f4e0000, 0x0f560000)
```

```
tenured generation total 10944K, used 0K [0x14560000, 0x15010000, 0x1f010000)
  the space 10944K, 0% used [0x14560000, 0x14560000, 0x14560200, 0x15010000)
compacting perm gen total 12288K, used 2056K [0x1f010000, 0x1fc10000, 0x23010000)
  the space 12288K, 16% used [0x1f010000, 0x1f2121d0, 0x1f212200, 0x1fc10000)
No shared spaces configured.
```

如果使用参数-XX:+UseParallelGC 配置 JVM，表示年轻代使用并行回收收集器，年老代使用串行收集器。输出如清单 7-14 所示。

代码清单 7-14 使用-XX:+UseParallelGC 参数

```
Heap
PSYoungGen total 4800K, used 1380K [0x1dac0000, 0x1e010000, 0x23010000)
 eden space 4160K, 33% used [0x1dac0000,0x1dc19130,0x1ded0000)
  from space 640K, 0% used [0x1df70000,0x1df70000,0x1e010000)
  to space 640K, 0% used [0x1ded0000,0x1ded0000,0x1df70000)
PSOldGen total 10944K, used 0K [0x13010000, 0x13ac0000, 0x1dac0000)
 object space 10944K, 0% used [0x13010000,0x13010000,0x13ac0000)
PSPermGen total 12288K, used 2056K [0x0f010000, 0x0fc10000, 0x13010000)
 object space 12288K, 16% used [0x0f010000,0x0f2121d0,0x0fc10000)
```

一次年老代串行回收器的工作输出如清单 7-15 所示。

代码清单 7-15 老年代串行回收器输出

```
[Full GC [Tenured: 1712K->1699K(10240K), 0.0071963 secs] 1712K->1699K(19456K),
[Perm : 377K->372K(12288K)], 0.0072393 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
```

上面的输出显示了垃圾回收前年老代和永久区的内存占用量，以及垃圾回收后年老代和永久区的内存使用量。

开启并行回收器可以使用参数-XX:+UseParNewGC，它表示年轻代使用并行收集器，年老代使用串行收集器。输出如清单 7-16 所示。

代码清单 7-16 使用-XX:+UseParNewGC 参数

```
[GC [ParNew: 825K->161K(4928K), 0.0155258 secs][Tenured: 8704K->661K(10944K),
0.0071964 secs] 9017K->661K(15872K), [Perm : 2049K->2049K(12288K)], 0.0228090 secs]
[Times: user=0.01 sys=0.00, real=0.01 secs]
Heap
par new generation total 4992K, used 179K [0x0f010000, 0x0f570000, 0x14560000)
 eden space 4480K, 4% used [0x0f010000, 0x0f03cda8, 0x0f470000)
  from space 512K, 0% used [0x0f470000, 0x0f470000, 0x0f4f0000)
  to space 512K, 0% used [0x0f4f0000, 0x0f4f0000, 0x0f570000)
tenured generation total 10944K, used 8853K [0x14560000, 0x15010000, 0x1f010000)
  the space 10944K, 80% used [0x14560000, 0x14e057c0, 0x14e05800, 0x15010000)
compacting perm gen total 12288K, used 2060K [0x1f010000, 0x1fc10000, 0x23010000)
  the space 12288K, 16% used [0x1f010000, 0x1f213228, 0x1f213400, 0x1fc10000)
No shared spaces configured.
```


7.3.3.5 Parallel 收集器

就目前而言, HotSpot 的新生代中除了拥有 ParNew 收集器是基于并行回收的以外, Parallel 收集器同样也采用了复制算法、并行回收和“Stop-the-World”机制。和 ParNew 收集器不同, Parallel 收集器可以控制程序的吞吐量大小, 因此它也被称为吞吐量优先的垃圾收集器。在程序中, 开发人员可以通过选项“-XX:GCTimeRatio”设置执行内存回收的时间所占 JVM 运行总时间的比例, 也就是控制 GC 的执行频率, 公式为 $1/(1+N)$, 默认值为 99, 也就是说, 将只有 1% 的时间用于执行垃圾回收。除此之外, Parallel 收集器还提供选项“-XX:MaxGCPauseMills”设置执行内存回收时“Stop-the-World”机制的暂停时间阈值, 如果指定了该选项, Parallel 收集器将会尽可能地在设定的时间范围内完成内存回收。

需要注意的是, 垃圾收集器中吞吐量和低延迟这两个目标本身是相互矛盾的, 因为如果选择以吞吐量优先, 那么必然需要降低内存回收的执行频率, 但是这样会导致 GC 需要更长的暂停时间来执行内存回授。相反的, 如果选择以低延迟优先为原则, 那么为了降低每次执行内存回收时的暂停时间, 也只能频繁地执行内存回收, 但这又引起了新生代内存的缩减和导致程序吞吐量的下降。举个例子, 在 60s 的 JVM 总运行时间里, 每次 GC 的执行频率是 20s/次, 那么 60s 内一共会执行 3 次内存回授, 按照每次 GC 耗时 100ms 来计算, 最终一共会有 300ms (3×100) 被用于执行垃圾回收。但是如果我们将选项“-XX:MaxGCPauseMills”的值调小后, 新生代的内存空间也会自动调整, 内存空间越小就越容易被耗尽, 也就越容易造成 GC 的执行频繁发生。之前在 60s 的 JVM 总运行时间里, 最终会有 300ms 被用于执行内存回收, 而如今 GC 的执行频率却是 10s/次, 60s 内将会执行 6 次内存回授, 按照每次 GC 耗时 60ms 来计算, 虽然看上去暂停时间更短了, 但最终会耗时 360ms (6×60) 用于执行内存回收, 很明显程序的吞吐量下降了。所以大家在设置这两个选项时, 一定需要注意控制在一个折中的范围之内。Parallel 收集器还提供一个“-XX:UseAdaptiveSizePolicy”选项用于设置 GC 的自动分代大小调节策略, 一旦设置这个选项后, 就意味着开发人员将不再需要显式地设置新生代中的一些细节参数, JVM 会根据自身的当前运行情况动态调整这些相关参数。

和 Serial 收集器一样, Parallel 收集器也提供用于执行老年代垃圾收集的 ParallelOld 收集器, Parallel Old 收集器采用了标记-压缩算法, 但同样也是基于并行回收和“Stop-the-World”机制。在程序吞吐量优先的应用场景中, Parallel 收集器和 Parallel Old 收集器的组合, 在 Server 模式下的内存回收性能很不错。在程序开发过程中, 开发人员可以通过选项“-XX:+UseParallelGC”手动指定使用 Parallel 收集器执行内存回收任务。

■ 年轻代并行回收(Parallel Scavenge)收集器

年轻代并行回收收集器也是使用复制算法的收集器。从表面上看, 它和并行收集器一样都是多线程、独占式的收集器。但是, 并行回收收集器有一个重要的特点: 它非常关注系统的吞吐量。

年轻代并行回收收集器可以使用以下参数启用:

-XX:+UseParallelGC:年轻代使用并行回收收集器, 年老代使用串行收集器。

-XX:+UseParallelOldGC:年轻代和年老代都是用并行回收收集器。

设定线程数量为 24 时, 运行输出如清单 7-17 所示。

代码清单 7-17 24 个线程的 Parallel 收集器

```

Heap
PSYoungGen      total 4800K, used 893K [0x1dac0000, 0x1e010000, 0x23010000)
  eden space 4160K, 21% used [0x1dac0000,0x1db9f570,0x1ded0000)
  from space 640K, 0% used [0x1df70000,0x1df70000,0x1e010000)
  to   space 640K, 0% used [0x1ded0000,0x1ded0000,0x1df70000)
ParOldGen        total 19200K, used 16384K [0x13010000, 0x142d0000, 0x1dac0000)
  object space 19200K, 85% used [0x13010000,0x14010020,0x142d0000)
PSPermGen        total 12288K, used 2054K [0x0f010000, 0x0fc10000, 0x13010000)
  object space 12288K, 16% used [0x0f010000,0x0f2119c0,0x0fc10000)
  
```

年轻代并行回收收集器可以使用以下参数启用：

-XX:+MaxGCPauseMills: 设置最大垃圾收集停顿时间，它的值是一个大于 0 的整数。收集器在工作时会调整 Java 堆大小或者其他一些参数，尽可能地把停顿时间控制在 MaxGCPauseMills 以内。如果希望减少停顿时间，而把这个值设置得很小，为了达到预期的停顿时间，JVM 可能会使用一个较小的堆(一个小堆比一个大堆回收快)，而这将导致垃圾回收变得很频繁，从而增加了垃圾回收总时间，降低了吞吐量。

-XX:+GCTimeRatio: 设置吞吐量大小，它的值是一个 0-100 之间的整数。假设 GCTimeRatio 的值为 n，那么系统将花费不超过 $1/(1+n)$ 的时间用于垃圾收集。比如 GCTimeRatio 等于 19，则系统用于垃圾收集的时间不超过 $1/(1+19)=5\%$ 。默认情况下，它的取值是 99，即不超过 1% 的时间用于垃圾收集。

除此之外，并行回收收集器与并行收集器另一个不同之处在于，它支持一种自适应的 GC 调节策略，使用 **-XX:+UseAdaptiveSizePolicy** 可以打开自适应 GC 策略。在这种模式下，年轻代的大小、eden 和 survivor 的比例、晋升年老代的对象年龄等参数会被自动调整，已达到在堆大小、吞吐量和停顿时间之间的平衡点。在手工调优比较困难的场合，可以直接使用这种自适应的方式，仅指定虚拟机的最大堆、目标的吞吐量(GCTimeRatio)和停顿时间(MaxGCPauseMills)，让虚拟机自己完成调优工作。

年轻代并行回收收集器的工作日志如清单 7-18 所示。

代码清单 7-18 年轻代并行回收收集器工作日志

```

Heap
PSYoungGen      total 4800K, used 893K [0x1dac0000, 0x1e010000, 0x23010000)
  eden space 4160K, 21% used [0x1dac0000,0x1db9f570,0x1ded0000)
  from space 640K, 0% used [0x1df70000,0x1df70000,0x1e010000)
  to   space 640K, 0% used [0x1ded0000,0x1ded0000,0x1df70000)
PSOldGen        total 19200K, used 16384K [0x13010000, 0x142d0000, 0x1dac0000)
  object space 19200K, 85% used [0x13010000,0x14010020,0x142d0000)
PSPermGen        total 12288K, used 2054K [0x0f010000, 0x0fc10000, 0x13010000)
  object space 12288K, 16% used [0x0f010000,0x0f2119c0,0x0fc10000)
  
```

上面的日志显示了回收前的内存大小和回收后的内存大小，以及花费的时间。

■ 年老代并行回收收集器

年老代的并行回收收集器也是一种多线程并发的收集器。和年轻代并行回收收集器一样，它也是一种关注吞吐量的收集器。年老代并行回收收集器使用标记-压缩算法，JDK1.6 之后开始启用。

使用-XX:+UseParallelOldGC 可以在年轻代和年老代都使用并行回收收集器，这是一对非常关注吞吐量的垃圾收集器组合，在对吞吐量敏感的系统，可以考虑使用。参数-XX:ParallelGCThreads 也可以用于设置垃圾回收时的线程数量。

设置线程数量为 100 时年老代并行回收收集器输出日志如清单 7-19 所示。

代码清单 7-19 年老代并行回收收集器线程 100 时日志

```
Heap
PSYoungGen      total 4800K, used 893K [0x1dac0000, 0x1e010000, 0x23010000)
 eden space 4160K, 21% used [0x1dac0000,0x1db9f570,0x1ded0000)
  from space 640K, 0% used [0x1df70000,0x1df70000,0x1e010000)
   to space 640K, 0% used [0x1ded0000,0x1ded0000,0x1df70000)
ParOldGen       total 19200K, used 16384K [0x13010000, 0x142d0000, 0x1dac0000)
 object space 19200K, 85% used [0x13010000,0x14010020,0x142d0000)
PSPermGen       total 12288K, used 2054K [0x0f010000, 0x0fc10000, 0x13010000)
 object space 12288K, 16% used [0x0f010000,0x0f2119c0,0x0fc10000)
```

7.3.3.6 CMS 收集器

CMS 全称是 Concurrent-Mark-Sweep。前面说过，在程序吞吐量优先的应用场景中，Parallel 收集器和 Parallel Old 收集器的组合，在 Server 模式下的内存回收性能很不错。但是在某些对系统响应速度要求比较高的项目中，大家总是希望系统能够快速做出响应，而不愿意看到过多的延迟。基于低延迟的考虑，JVM 的设计者们提供了基于并行回收的 CMS（Concurrent-Marking-Sweep）收集器，它是一款优秀的老年代垃圾收集器，也可以被称作 Mostly-Concurrent 收集器。CMS 天生为并发而生，低延迟是它的优势，不过垃圾收集算法却并没有采用标记-复制算法，而是采用标记-清除算法，并且也会因为“Stop-the-world”机制而出现短暂的暂停。

CMS 的执行过程可以分为 4 个主要阶段，即初始标记（Initial-Mark）阶段、并发标记（Concurrent-Mark）阶段、再次标记（Remark）阶段和并发清除（Concurrent-Sweep）阶段。

CMS 收集器的回收周期以一个称之为初始标记的阶段开始，在这个阶段中，程序中所有的工作线程都将会因为“Stop-the-World”机制而出现短暂的暂停，这个阶段的主要任务就是标记出内存中那些被根对象集合所连接的目标对象是否可达，一旦标记完成之后就会恢复之前被暂停的所有应用线程。接下来将会进入并发标记阶段，而这个阶段的主要任务就是将之前的不可达对象标记为垃圾对象。在 CMS 最终执行内存回收之前，尽管看上去这些垃圾对象都已经被成功标记了，但是由于在并发标记阶段中，程序的工作线程会和垃圾收集线程同时运行或者交叉运行，因此在并发标记阶段将无法有效确保之前被标记为垃圾的无用对象的引用关系遭到更改，为了解决这个问题，CMS 会进入到再次标记阶段，这样一来，程序会因为“Stop-the-World”机制而再次出现短暂的暂停，以确保这些垃圾对象都能够被成功且正确地标记。当经历过初始标记、并发标记和再次标记这三个阶段后，CMS 最终将会进入到并发清除阶段执行内存回收，释放掉无用对象所占用的

的内存空间。

尽管 CMS 收集器采用的是并行回收,但是在其初始化标记和再次标记这两个阶段中仍然需要执行“Stop-the-World”机制暂停程序中的工作线程,不过暂停时间并不会太长,因此可以说明目前所有的垃圾收集器都做不到完全不需要“Stop-the-World”,只是尽可能地降低暂停时间。

之前介绍的几款老年代垃圾收集器,比如 Serial Old 收集器、Parallel Old 收集器的垃圾收集算法都是采用标记-压缩来避免执行 Full GC 后产生内存碎片,而 CMS 收集器的垃圾收集算法采用的是标记-清除算法,这意味着每次执行完内存回收后,由于被执行内存回收的无用对象所占用的内存空间极有可能是非连续的一些内存块,不可避免地将会产生一些内存碎片。那么 CMS 在为新对象分配内存空间后,将无法使用指针碰撞(Bump the Pointer)技术,而只能选择空闲列表(Free List)执行内存分配。

在 HotSpot 中,当垃圾收集器执行完内存回收后,如果内存空间中产生内存碎片,那么则只能选择空闲列表作为内存分配算法为新对象分配内存空间。简单来说,会有 JVM 负责维护一个列表,其中所记录的内容就是当前内存空间中可用内存块的坐标,当执行内存分配时,会从列表中定位到一个与新对象所需内存大小一致的连续内存块用于存储生成的对象实例。考虑到内存碎片存在的弊端,CMS 收集器提供选项“-XX:+UseCMS-CompactAtFullCollection”,用于指定在执行完 Full GC 后是否对内存空间进行压缩整理,以此避免内存碎片的产生。不过由于内存压缩整理过程无法并发执行,所带来的问题就是停顿时间变得更长了,因此 CMS 收集器还提供另外一个选项“-XX:CMSFullGCs-BeforeCompaction”,用于设置在执行多少次 Full GC 后对内存空间进行压缩整理。除了会产生内存碎片外,CMS 收集器还存在一个不容忽视的问题,那就是在并发标记阶段由于程序的工作线程和垃圾收集线程是同时运行或者交叉运行的,那么在并发标记阶段如果产生新的垃圾对象,CMS 将无法对这些垃圾对象进行标记,最终会导致这些新产生的垃圾对象没有被及时回收,从而只能在下一次执行 GC 时释放这些之前未被回收的内存空间。

尽管 Full GC 大多数时候只会发生在老年代垃圾回收阶段,但是实际上 Full GC 的回收范围却不单单仅限于老年代中,从严格意义上来说,Full GC 的回收范围几乎覆盖了整个堆空间,因此 Full GC 将会比 Minor GC 耗费更长的时间来完成垃圾收集。在 HotSpot 中,除了 CMS 收集器之外的任何老年代垃圾收集器在执行内存回收时,都将会执行 Full GC,只有 G1 收集器较为特殊。CMS 收集器提供选项“-XX:CMSInitiatingOccupancyFraction”用于设置当老年代中的内存使用率达到多少百分比的时候执行内存回收(低版本的 JDK 默认值为 68%,JDK6 及以上版本默认值为 92%),这里的内存回收范围仅限于老年代,而非整个堆空间,因此通过该选项便可以有效降低 Full GC 的执行次数。当然并不是说使用了 CMS 收集器之后,就永远不会再触发 Full GC 了,一旦 CMS 在执行过程中出现“Promotion Failed”或“Concurrent Mode Failure”时,将仍然有可能会触发 Full GC 操作。在程序开发过程中,开发人员可以通过选项“-XX:+UseConcMarkSweepGC”来手动指定使用 CMS 收集器执行内存回收任务。

CMS 收集器在其主要的工作阶段虽然没有暴力地彻底暂停应用程序线程,但是由于它和应用程序线程并发执行,相互抢占 CPU,所以在 CMS 执行期内对应用程序吞吐量造成一定影响。CMS 默认启动的线程数是 $(ParallelGCThreads+3)/4$,ParallelGCThreads 是年轻代并行收集器的线程数,也可以通过-XX:ParallelCMSThreads 参数手工设定 CMS 的线程数量。当 CPU 资源比较紧张时,受到 CMS 收集器线程的影响,应用程序的性能在垃圾回收阶段可能会非常糟糕。

由于 CMS 收集器不是独占式的回收器，在 CMS 回收过程中，应用程序仍然在不停地工作。在应用程序工作过程中，又会不断地产生垃圾。这些新生成的垃圾在当前 CMS 回收过程中是无法清除的。同时，因为应用程序没有中断，所以在 CMS 回收过程中，还应该确保应用程序有足够的内存可用。因此，CMS 收集器不会等待堆内存饱和时才进行垃圾回收，而是当前堆内存使用率达到某一阈值时，便开始进行回收，以确保应用程序在 CMS 工作过程中依然有足够的空间支持应用程序运行。

这个回收阈值可以使用-XX:CMSInitiatingOccupancyFraction 来指定，默认是 68。即当年老代的空间使用率达到 68%时，会执行一次 CMS 回收。如果应用程序的内存使用率增长很快，在 CMS 的执行过程中，已经出现了内存不足的情况，此时，CMS 回收将会失败，JVM 将启动年老代串行收集器进行垃圾回收。如果这样，应用程序将完全中断，直到垃圾收集完成，这是，应用程序的停顿时间可能很长。因此，根据应用程序的特点，可以对-XX:CMSInitiatingOccupancyFraction 进行调优。如果内存增长缓慢，则可以设置一个稍大的值，大的阈值可以有效降低 CMS 的触发频率，减少年老代回收的次数可以较为明显地改善应用程序性能。反之，如果应用程序内存使用率增长很快，则应该降低这个阈值，以避免频繁触发年老代串行收集器。

标记-清除算法将会造成大量内存碎片，离散的可用空间无法分配较大的对象。在这种情况下，即使堆内存仍然有较大的剩余空间，也可能被迫进行一次垃圾回收，以换取一块可用的连续内存，这种现象对系统性能是相当不利的，为了解决这个问题，CMS 收集器还提供了几个用于内存压缩整理的算法。

-XX:+UseCMSCompactAtFullCollection 开发可以使 CMS 在垃圾收集完成后，进行一次内存碎片整理。内存碎片的整理并不是并发进行的。-XX:CMSFullGCsBeforeCompaction 参数可以用于设定进行多少次 CMS 回收后，进行一次内存压缩。

-XX:CMSInitiatingOccupancyFractio 设置为 100，设置-XX:+UseCMSCompactAtFullCollection，设置-XX:CMSFullGCsBeforeCompaction，日志输出如清单 7-20 所示。

代码清单 7-20 线程 100 个时使用 CMS 收集器日志输出

```
[GC [DefNew: 825K->149K(4928K), 0.0023384 secs][Tenured: 8704K->661K(10944K),
0.0587725 secs] 9017K->661K(15872K), [Perm : 374K->374K(12288K)], 0.0612037 secs]
[Times: user=0.01 sys=0.02, real=0.06 secs]
Heap
def new generation      total 4992K, used 179K [0x27010000, 0x27570000, 0x2c560000)
  eden space 4480K,      4% used [0x27010000, 0x2703cda8, 0x27470000)
  from space 512K,       0% used [0x27470000, 0x27470000, 0x274f0000)
  to   space 512K,       0% used [0x274f0000, 0x274f0000, 0x27570000)
tenured generation      total 10944K, used 8853K [0x2c560000, 0x2d010000, 0x37010000)
  the space 10944K,      80% used [0x2c560000, 0x2ce057c8, 0x2ce05800, 0x2d010000)
compacting perm gen     total 12288K, used 374K [0x37010000, 0x37c10000, 0x3b010000)
  the space 12288K,      3% used [0x37010000, 0x3706db10, 0x3706dc00, 0x37c10000)
  ro space 10240K,       51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
  rw space 12288K,       55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

如果使用参数-XX:+UseConcMarkSweepGC，表示年轻代使用并行收集器，年老代使用 CMS。输出如清单 7-21 所示。

代码清单 7-21 使用-XX:+UseConcMarkSweepGC 参数

```
[GC [ParNew: 8967K->669K(14784K), 0.0040895 secs] 8967K->669K(63936K), 0.0043255
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
  par new generation   total 14784K, used 9389K [0x03f50000, 0x04f50000, 0x04f50000)
    eden space 13184K,  66% used [0x03f50000, 0x047d3e58, 0x04c30000)
    from space 1600K,  41% used [0x04dc0000, 0x04e67738, 0x04f50000)
    to   space 1600K,   0% used [0x04c30000, 0x04c30000, 0x04dc0000)
  concurrent mark-sweep generation total 49152K, used 0K [0x04f50000, 0x07f50000,
0x09f50000)
    concurrent-mark-sweep perm gen total 12288K, used 2060K [0x09f50000, 0x0ab50000,
0x0df50000)
```

并行收集器工作时的线程数量可以使用-XX:ParallelGCThreads 参数指定。一般，最好与 CPU 数量相当，避免过多的线程数影响垃圾收集性能。在默认情况下，当 CPU 数量小于 8 个，ParallelGCThreads 的值等于 CPU 数量，大于 8 个 ParallelGCThreads 的值等于 $3 + [5 * CPU_Count] / 8$ 。设置为 8 个线程后输出如清单 7-22 所示。

代码清单 7-22 并发收集器的线程设置 8 个时日志输出

```
[GC [ParNew: 8967K->676K(14784K), 0.0036983 secs] 8967K->676K(63936K), 0.0037662
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
  par new generation   total 14784K, used 9395K [0x040e0000, 0x050e0000, 0x050e0000)
    eden space 13184K,  66% used [0x040e0000, 0x04963e58, 0x04dc0000)
    from space 1600K,  42% used [0x04f50000, 0x04ff9100, 0x050e0000)
    to   space 1600K,   0% used [0x04dc0000, 0x04dc0000, 0x04f50000)
  concurrent mark-sweep generation total 49152K, used 0K [0x050e0000, 0x080e0000,
0x0a0e0000)
    concurrent-mark-sweep perm gen total 12288K, used 2060K [0x0a0e0000, 0x0ace0000,
0x0e0e0000)
```

并发收集器的线程设置 128 个时日志输出

```
[GC [ParNew: 8967K->664K(14784K), 0.0207327 secs] 8967K->664K(63936K), 0.0208077
secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
```

并发收集器的线程设置 640 个时日志输出

```
[GC [ParNew: 8967K->667K(14784K), 0.2323704 secs] 8967K->667K(63936K), 0.2324778
secs] [Times: user=0.34 sys=0.02, real=0.23 secs]
```

并发收集器的线程设置 1280 个时日志输出

```
Error occurred during initialization of VM
Too small new size specified
```

7.3.3.7 G1 收集器

G1 全称是 Garbage-First。The Garbage-First (G1)垃圾收集器完全支持在 Oracle JDK 7 Update 4 和以后的版本。G1 收集器是一个服务器类型的垃圾收集器，目标用于大存储器的多处理器机器。

它符合垃圾收集 (GC) 以很高的概率, 暂停时间的目标, 同时实现高吞吐量。整个堆的操作, 比如全局标记, 同时执行的应用程序线程。这可以防止中断堆或实时数据的大小比例。

G1 收集器的设计初衷是为了替代 CMS 收集器而生, 自身的目标是以更高的计算成本为代价最小化 STW 中断时间。G1 更适合于低延迟应用程序, 如 Web 服务器简单来说, G1 是一款基于并行和并发、低延迟以及暂停时间更加可控的区域化分代式垃圾收集器。G1 在内存空间的设计上重新塑造了整个 Java 堆区, 尽管同样也是基于分代的概念执行内存分配和垃圾回收, 但是 G1 却并没有采用传统物理隔离的新生代和老年代布局方式, 仅仅逻辑上划分为新生代和老年代, 它选择的将 Java 堆区划分成 2048 个大小相同的独立 Region 块, 每个 Region 块之间可能是不连续的, 其大小根据堆空间的实际大小而定, 整体被控制在 1MB 到 32MB 之间。这样划分的好处在于可以更好地提升 GC 的回收频率和缩短 “Stop-the-World” 机制的暂停时间以换取更大的程序吞吐量, 甚至能够真正地精确控制程序的暂停时间等。这是因为 G1 收集器在执行内存回收时, 会优先释放掉整个 Java 堆区中一些占用内存较大的 Region 块, 从而可以避免像以往一样直接扫描整个 Java 堆区 (如果一个 Region 块中引用另一个 Region 块内的对象时, 则通过 Remembered Set 技术避免全堆扫描)。至于执行内存回收时 “Stop-the-World” 机制的暂停时间更加可控, 是相对于 Parallel 收集器而言, 尽管 Parallel 收集器也提供选项 “-XX:MaxGCPauseMillis” 控制程序的暂停时间, 但是在内存空间释放操作中, 暂停时间其实并非是完全可控的, 只能说尽可能控制在预期范围内, 但谁也无法保证内存回收时所带来的暂停时间一定会百分百地控制在规定的阈值内, 所以一定会有些误差。而由于 G1 收集器并非全堆扫描, 只优先回收占用内存较大的一些 Region 块, 所以 G1 收集器的暂停时间会更加可控。

前面说过, 堆被划分成多个大小相等的堆区, 每一个连续范围的虚拟内存。G1 执行并行全局标记阶段确定堆上对象的活跃度。标记阶段完成后, G1 知道哪个区域大多是空的。它先收集在这些区域, 通常会产生大量的自由空间。这就是为什么这种垃圾收集方法叫作 G1 收集器。顾名思义, G1 集中收集和压缩活动区域的堆, 类似可以回收的对象。G1 使用暂停预测模型来满足用户定义的暂停时间目标和选择一些区域收集基于指定的暂停时间目标。通过 G1 作为成熟的回收使用垃圾收集疏散区域。G1 的副本从一个或一个以上的堆区对象单一区域堆上, 并在这一过程中双方的契约和释放内存。总之就是多处理器, 减少暂停时间和增加吞吐量。因此, 每个垃圾收集, G1 连续工作来减少碎片, 在用户定义的暂停时间工作。这是超越的以往方法的能力。CMS (并发标记扫描) 垃圾收集不进行压缩。parallelold 垃圾收集仅执行整个堆压缩, 从而导致了相当的暂停时间。我们需要注意, G1 进行的收集方式属于非实时收集, 即便符合所有的回收标准, 我们还是不能百分百地确定会被回收。根据统计的数据显示, G1 也会事先对用户的制定目标时间内的收集目标进行统计, 尽量多地回收垃圾。因此, 具有一个合理准确的收集区域的成本模型, 并使用该模式来确定哪一个多少区域要收集当目标停留的暂停时间内。

G1 收集器的执行过程主要可以划分为 6 个阶段, 即初始标记 (Initial-Mark) 阶段、根区域扫描 (Root-Region-Scanning) 阶段、并发标记 (Concurrent-Marking) 阶段、再次标记 (Remark) 阶段、清除 (Cleanup) 阶段和拷贝 (Copying) 阶段。

G1 收集器的执行过程与 CMS 收集器相似。在初始标记阶段中, 程序中所有的工作线程都会因为 “Stop-the-World” 机制而出现短暂的暂停, 该阶段的主要任务是标记 Root-Region。一旦标记完成之后就会恢复之前被暂停的所有应用线程。接下来将会进入到根区域扫描阶段, 该阶段的主要任务是扫描 Root-Region 中引用老年代的一些 Region 块, 只有在执行完扫描之后, 才能开始

下一次新生代内存回收。

并发标记阶段的主要任务是找出整个 Java 堆区中的存活对象，由于该阶段并不会导致程序出现暂停，因此在执行的过程中允许被新生代内存回收打断。再次标记阶段和初始标记阶段同样都是基于“Stop-the-World”机制的，该阶段的主要任务就是完成整个堆区中存活对象的标记。清除标记阶段由三部分组成，首先会计算出所有的活跃对象并完全释放一些自由的 Region 块，然后处理 Remembered Set，再次需要注意的是，这两部分操作将会暂停程序中的应用线程，然后并发重置空闲的一些 Region 块，并将它们放回至空闲列表中。最后的拷贝阶段也是基于“Stop-the-World”机制的，该阶段的主要任务就是将存活对象复制到未使用过的 Region 块中。当经历过这 6 个阶段后，G1 收集器的内存回收任务就算是完成了。在程序开发过程中，程序员可以通过选项“-XX:UseG1GC”来手动指定使用 G1 收集器执行内存回收任务。

由于 G1 主要关注于多 CPU 多线程，所以内存分配采用 thread-local allocation buffers(TLABs) 技术。每个分配线程都有一个自己的 buffers 用来分配对象，当 buffers 用完或者不够的时候，去重新申请一块内存放在自己的 thread-local 里面。这样对象的内存分配被最小化到私有的 buffers 里面，缓解了并发分配内存的压力。

当 region 被填满后，分配内存的线程会重新选择一个新的 region。空 region 被组织到一个 linked list 里面，这样可以快速找到新的 region。

对于大对象的分配不是在 TLABs 进行的，而是在 TLABs 之外。当一个对象的大小超过 region 的 3/4 的时候，这个对象被认为是巨大的(humongous)。巨大的对象被分配到特殊的区域(heap regions)。这些区域只包含巨大对象(humongous object)。

G1 收集器的目标是作为一款服务器的垃圾收集器，因此，它在吞吐量和停顿控制上，预期要优于 CMS 收集器。

与 CMS 收集器相比，G1 收集器是基于标记-压缩算法的。因此，它不会产生空间碎片，也没有必要在收集完成后，进行一次独占式的碎片整理工作。G1 收集器还可以进行非常精确的停顿控制。它可以让开发人员指定在长度为 M 的时间段中，垃圾回收时间不超过 N。使用参数-XX:+UnlockExperimentalVMOptions -XX:+UseG1GC 来启用 G1 回收器，设置 G1 回收器的目标停顿时间：-XX:MaxGCPauseMills=20,-XX:GCPauseIntervalMills=200。

总体来说，G1 跟 CMS 一样，是一块低延时的收集器，同样牺牲了吞吐量，不过二者之间得到了很好的权衡。G1 与 CMS 对比有以下不同点：

- (1) **分代**：CMS 中，堆被分为 PermGen, YoungGen, OldGen；而 YoungGen 又分了两个 survivo 区域。在 G1 中，堆被平均分成几个区域(region)，在每个区域中，虽然也保留了新老代的概念，但是收集器是以整个区域为单位收集的。
- (2) **算法**：相对于 CMS 的“标记——清理”算法，G1 会使用压缩算法，保证不产生多余的碎片。收集阶段，G1 会将某个区域存活的对象拷贝的其他区域，然后将整个区域回收。
- (3) **停顿时间可控**：为了缩短停顿时间，G1 建立可预存停顿模型，这样在用户设置的停顿时间范围内，G1 会选择适当的区域进行收集，确保停顿时间不超过用户指定时间。

7.3.3.8 收集器对系统性能的影响

上面说了这么多关于各个垃圾收集器的理论、参数、优缺点，我们现在通过一个示例来应用不同的垃圾回收器，让我们来看看不同的效果。

示例程序中通过对 7-23 所示的代码运行 1 万次循环，每次分配 512*100B 空间，采用不同的垃圾回收器，输出程序运行所消耗的时间。

代码清单 7-23 示例程序代码

```
import java.util.HashMap;

public class GCTest {
    static HashMap map = new HashMap();

    public static void main(String[] args) {
        long begintime = System.currentTimeMillis();
        for(int i=0;i<10000;i++){
            if(map.size()*512/1024/1024>=400){
                map.clear();//保护内存不溢出
                System.out.println("clean map");
            }
            byte[] b1;
            for(int j=0;j<100;j++){
                b1 = new byte[512];
                map.put(System.nanoTime(), b1);//不断消耗内存
            }
        }
        long endtime = System.currentTimeMillis();
        System.out.println(endtime-begintime);
    }
}
```

使用参数-Xmx512M -Xms512M -XX:+UseParNewGC 运行代码，输出如清单 7-24 所示。

代码清单 7-24 使用 ParNewGC 收集器运行日志

```
clean map 8565
cost time=1655
```

使用参数-Xmx512M -Xms512M -XX:+UseParallelOldGC -XX:ParallelGCThreads=8 运行代码，输出如清单 7-25 所示。

代码清单 7-25 使用 ParallelOld 收集器运行日志

```
clean map 8798
cost time=1998
```

使用参数-Xmx512M -Xms512M -XX:+UseSerialGC 运行代码，输出如清单 7-26 所示。

代码清单 7-26 使用 Serial 收集器运行日志

```
clean map 8864
cost time=1717
```

使用参数-Xmx512M -Xms512M -XX:+UseConcMarkSweepGC 运行代码，输出如清单 7-27 所示。

代码清单 7-27 使用 CMS 收集器运行日志

```
clean map 8862
cost time=1530
```

根据 Oracle 公司官方针对 Java9 的通报，G1 将取代并行垃圾收集器成为服务器配置的默认选项。正如 Oracle 在内存管理白皮书中描述的那样，并行垃圾收集器的设计初衷，是通过不常发生（但可能时间比较长）的 Stop-The-World（STW）中断最大化应用程序吞吐量。并行垃圾收集器将消耗的总计算时间最小化，长远来看，其破坏性更小，因此可以提供更好的整体性能。该收集器非常适合对响应时间要求不高的应用程序，比如批处理。

7.4 实用 JVM 实验

由于 Java 字节码是运行在 JVM 虚拟机上的，同样的字节码使用不同的 JVM 虚拟机参数运行，其性能表现可能各不一样。为了能使系统性能最优，就需要选择使用合适的 JVM 参数运行 Java 应用程序，我们本节就针对不同的 JVM 参数进行实验讲解。

7.4.1 将新对象预留在年轻代

由于 Full GC 的成本远远高于 Minor GC，因此尽可能将对象分配在年轻代是一项明智的做法。虽然在大部分情况下，JVM 会尝试在 eden 区分配对象，但是由于空间紧张等问题，很可能不得不将部分年轻对象提前向年老代压缩。因此，在 JVM 参数调优时可以为应用程序分配一个合理的年轻代空间，以最大限度避免新对象直接进入年老代的情况发生。

代码清单 7-28 所示代码分配了 4MB 内存空间，观察一下它的内存使用情况。

代码清单 7-28 示例程序代码

```
public class PutInEden {
    public static void main(String[] args){
        byte[] b1,b2,b3,b4;//定义变量
        b1=new byte[1024*1024];//分配 1MB 堆空间，考察堆空间的使用情况
        b2=new byte[1024*1024];
        b3=new byte[1024*1024];
        b4=new byte[1024*1024];
    }
}
```

使用 JVM 参数-XX:+PrintGCDetails -Xmx20M -Xms20M 运行上例，输出如清单 7-29 所示：

代码清单 7-29 示例程序代码运行输出

```
[GC [DefNew: 5504K->640K(6144K), 0.0114236 secs] 5504K->5352K(19840K), 0.0114595
secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
[GC [DefNew: 6144K->640K(6144K), 0.0131261 secs] 10856K->10782K(19840K), 0.0131612
secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
[GC [DefNew: 6144K->6144K(6144K), 0.0000170 secs] [Tenured: 10142K->13695K(13696K),
0.1069249 secs] 16286K->15966K(19840K), [Perm : 376K->376K(12288K)], 0.1070058 secs]
[Times: user=0.03 sys=0.00, real=0.11 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0302067 secs] 19839K->19595K(19840K),
[Perm : 376K->376K(12288K)], 0.0302635 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0311986 secs] 19839K->19839K(19840K),
[Perm : 376K->376K(12288K)], 0.0312515 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0358821 secs] 19839K->19825K(19840K),
[Perm : 376K->371K(12288K)], 0.0359315 secs] [Times: user=0.05 sys=0.00, real=0.05 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0283080 secs] 19839K->19839K(19840K),
[Perm : 371K->371K(12288K)], 0.0283723 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0284469 secs] 19839K->19839K(19840K),
[Perm : 371K->371K(12288K)], 0.0284990 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0283005 secs] 19839K->19839K(19840K),
[Perm : 371K->371K(12288K)], 0.0283475 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0287757 secs] 19839K->19839K(19840K),
[Perm : 371K->371K(12288K)], 0.0288294 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0288219 secs] 19839K->19839K(19840K),
[Perm : 371K->371K(12288K)], 0.0288709 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0293071 secs] 19839K->19839K(19840K),
[Perm : 371K->371K(12288K)], 0.0293607 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0356141 secs] 19839K->19838K(19840K),
[Perm : 371K->371K(12288K)], 0.0356654 secs] [Times: user=0.01 sys=0.00, real=0.03 secs]
Heap
 def new generation   total 6144K, used 6143K [0x35c10000, 0x362b0000, 0x362b0000)
   eden space 5504K, 100% used [0x35c10000, 0x36170000, 0x36170000)
   from space 640K,  99% used [0x36170000, 0x3620fc80, 0x36210000)
   to   space 640K,   0% used [0x36210000, 0x36210000, 0x362b0000)
 tenured generation    total 13696K, used 13695K [0x362b0000, 0x37010000,
0x37010000)
   the space 13696K,  99% used [0x362b0000, 0x3700fff8, 0x37010000, 0x37010000)
 compacting perm gen  total 12288K, used 371K [0x37010000, 0x37c10000, 0x3b010000)
   the space 12288K,   3% used [0x37010000, 0x3706cd20, 0x3706ce00, 0x37c10000)
    ro space 10240K,  51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
    rw space 12288K,  55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

上面的输出显示年轻代 eden 的大小有 5MB 左右。分配足够大的年轻代空间，使用 jvm 参数 -XX:+PrintGCDetails -Xmx20M -Xms20M -Xmn6M 运行上例，输出如 7-30 所示。

代码清单 7-30 示例程序代码运行输出

```
[GC [DefNew: 4992K->576K(5568K), 0.0116036 secs] 4992K->4829K(19904K), 0.0116439
secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
[GC [DefNew: 5568K->576K(5568K), 0.0130929 secs] 9821K->9653K(19904K), 0.0131336
```



```
secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
[GC [DefNew: 5568K->575K(5568K), 0.0154148 secs] 14645K->14500K(19904K), 0.0154531
secs] [Times: user=0.00 sys=0.01, real=0.01 secs]
[GC [DefNew: 5567K->5567K(5568K), 0.0000197 secs] [Tenured: 13924K->14335K(14336K),
0.0330724 secs] 19492K->19265K(19904K), [Perm : 376K->376K(12288K)], 0.0331624 secs]
[Times: user=0.03 sys=0.00, real=0.03 secs]
[Full GC [Tenured: 14335K->14335K(14336K), 0.0292459 secs] 19903K->19902K(19904K),
[Perm : 376K->376K(12288K)], 0.0293000 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
[Full GC [Tenured: 14335K->14335K(14336K), 0.0278675 secs] 19903K->19903K(19904K),
[Perm : 376K->376K(12288K)], 0.0279215 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
[Full GC [Tenured: 14335K->14335K(14336K), 0.0348408 secs] 19903K->19889K(19904K),
[Perm : 376K->371K(12288K)], 0.0348945 secs] [Times: user=0.05 sys=0.00, real=0.05 secs]
[Full GC [Tenured: 14335K->14335K(14336K), 0.0299813 secs] 19903K->19903K(19904K),
[Perm : 371K->371K(12288K)], 0.0300349 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]
[Full GC [Tenured: 14335K->14335K(14336K), 0.0298178 secs] 19903K->19903K(19904K),
[Perm : 371K->371K(12288K)], 0.0298688 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space[Full GC
[Tenured: 14335K->14335K(14336K), 0.0294953 secs] 19903K->19903K(19904K), [Perm :
371K->371K(12288K)], 0.0295474 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
[Full GC [Tenured
: 14335K->14335K(14336K), 0.0287742 secs] 19903K->19903K(19904K), [Perm :
371K->371K(12288K)], 0.0288239 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
[Full GC [Tenured at GCTestTimeTest.main(GCTestTimeTest.java:16)
: 14335K->14335K(14336K), 0.0287102 secs] 19903K->19903K(19904K), [Perm :
371K->371K(12288K)], 0.0287627 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
Heap
def new generation total 5568K, used 5567K [0x35c10000, 0x36210000, 0x36210000)
eden space 4992K, 100% used [0x35c10000, 0x360f0000, 0x360f0000)
from space 576K, 99% used [0x36180000, 0x3620ffe8, 0x36210000)
to space 576K, 0% used [0x360f0000, 0x360f0000, 0x36180000)
tenured generation total 14336K, used 14335K [0x36210000, 0x37010000,
0x37010000)
the space 14336K, 99% used [0x36210000, 0x3700ffd8, 0x37010000, 0x37010000)
compacting perm gen total 12288K, used 371K [0x37010000, 0x37c10000, 0x3b010000)
the space 12288K, 3% used [0x37010000, 0x3706ce28, 0x3706d000, 0x37c10000)
ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

通过设置一个较大的年轻代预留新对象,设置合理的 survivor 区并且提供 survivor 区的使用率,可以将年轻对象保存在年轻代。一般来说, survivor 区的空间不够,或者占用量达到 50%时,就会将对象进入年老代(不管它的年龄有多大)。代码清单 7-31 所示的程序中我们尝试让对象 b3 可以被回收。

代码清单 7-31 示例程序代码

```
public class PutInEden2 {
    public static void main(String[] args){
        byte[] b1,b2,b3;
```



```

b1=new byte[1024*512]; //分配 0.5MB 堆空间
b2=new byte[1024*1024*4]; //分配 4MB 堆空间
b3=new byte[1024*1024*4];
b3=null; //使 b3 可以被回收
b3=new byte[1024*1024*4]; //分配 4MB 堆空间
}
}

```

使用参数-XX:+PrintGCDetails -Xmx1000M -Xms500M -Xmn100M -XX:SurvivorRatio=8 运行上例，输出如清单 7-32 所示。

代码清单 7-32 示例程序代码运行输出

```

Heap
def new generation total 92160K, used 11878K [0x0f010000, 0x15410000, 0x15410000)
eden space 81920K, 2% used [0x0f010000, 0x0f1a9a20, 0x14010000)
from space 10240K, 99% used [0x14a10000, 0x1540fff8, 0x15410000)
to space 10240K, 0% used [0x14010000, 0x14010000, 0x14a10000)
tenured generation total 409600K, used 86434K [0x15410000, 0x2e410000,
0x4d810000)
the space 409600K, 21% used [0x15410000, 0x1a878b18, 0x1a878c00, 0x2e410000)
compacting perm gen total 12288K, used 2062K [0x4d810000, 0x4e410000, 0x51810000)
the space 12288K, 16% used [0x4d810000, 0x4da13b18, 0x4da13c00, 0x4e410000)
No shared spaces configured.

```

年轻代分配了 8M，年老代也分配了 8M。加上-XX:TargetSurvivorRatio=90 参数后，可以提高 from 区的利用率，使 from 区使用到 90%时，再将对象送入年老代，输出如清单 7-33 所示。

代码清单 7-33 示例程序代码运行输出

```

Heap
def new generation total 9216K, used 9215K [0x35c10000, 0x36610000, 0x36610000)
eden space 8192K, 100% used [0x35c10000, 0x36410000, 0x36410000)
from space 1024K, 99% used [0x36510000, 0x3660fc50, 0x36610000)
to space 1024K, 0% used [0x36410000, 0x36410000, 0x36510000)
tenured generation total 10240K, used 10239K [0x36610000, 0x37010000,
0x37010000)
the space 10240K, 99% used [0x36610000, 0x3700ff70, 0x37010000, 0x37010000)
compacting perm gen total 12288K, used 371K [0x37010000, 0x37c10000, 0x3b010000)
the space 12288K, 3% used [0x37010000, 0x3706cd90, 0x3706ce00, 0x37c10000)
ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)

```

将 SurvivorRatio 设置为 2，将 B1 对象预存在年轻代。输出如清单 7-34 所示。

代码清单 7-34 示例程序代码

```

Heap
def new generation total 7680K, used 7679K [0x35c10000, 0x36610000, 0x36610000)
eden space 5120K, 100% used [0x35c10000, 0x36110000, 0x36110000)
from space 2560K, 99% used [0x36110000, 0x3638fff0, 0x36390000)

```

```
to space 2560K, 0% used [0x36390000, 0x36390000, 0x36610000)
tenured generation total 10240K, used 10239K [0x36610000, 0x37010000,
0x37010000)
the space 10240K, 99% used [0x36610000, 0x3700fff0, 0x37010000, 0x37010000)
compacting perm gen total 12288K, used 371K [0x37010000, 0x37c10000, 0x3b010000)
the space 12288K, 3% used [0x37010000, 0x3706ce28, 0x3706d000, 0x37c10000)
ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

7.4.2 大对象进入年老代

在大部分情况下，将对象分配在年轻代是合理的。但是，对于大对象，因为大对象出现在年轻代很可能扰乱年轻代 GC，并破坏年轻代原有的对象结构。因为尝试在年轻代分配大对象，很可能导致空间不足，为了有足够的空间容纳大对象，JVM 不得不将年轻代中的年轻对象挪到年老代。因为大对象占用空间多，所以可能需要移动大量小的年轻对象进入年老代，这对 GC 相当不利。

基于以上原因，可以将大对象直接分配到年老代，保持年轻代对象结构的完整性，这样可以提高 GC 的效率。如果一个大对象同时又是一个短命的对象，假设这种情况出现很频繁，那对于 GC 来说会是一场灾难。原本应该用于存放永久对象的年老代，被短命的对象塞满，这也意味着对堆空间进行了洗牌，扰乱了分代内存回收的基本思路。因此，在软件开发过程中，应该尽可能避免使用短命的大对象。

可以使用参数-XX:PetenureSizeThreshold 设置大对象直接进入年老代的阈值。当对象的大小超过这个值时，将直接在年老代分配。参数-XX:PetenureSizeThreshold 只对串行收集器和年轻代并行收集器有效，并行回收收集器不识别这个参数。

代码清单 7-35 PetenureSizeThreshold 参数示例程序代码

```
public class BigObj20ld {
    public static void main(String[] args){
        byte[] b;
        b = new byte[1024*1024]; //分配一个 1MB 的对象
    }
}
```

使用 JVM 参数-XX:+PrintGCDetails -Xmx20M -Xms20MB 运行，可以得到清单 7-36 所示输出。

代码清单 7-36 示例程序代码运行输出

```
Heap
def new generation total 6144K, used 1378K [0x35c10000, 0x362b0000, 0x362b0000)
eden space 5504K, 25% used [0x35c10000, 0x35d689e8, 0x36170000)
from space 640K, 0% used [0x36170000, 0x36170000, 0x36210000)
to space 640K, 0% used [0x36210000, 0x36210000, 0x362b0000)
tenured generation total 13696K, used 0K [0x362b0000, 0x37010000, 0x37010000)
the space 13696K, 0% used [0x362b0000, 0x362b0000, 0x362b0200, 0x37010000)
compacting perm gen total 12288K, used 374K [0x37010000, 0x37c10000, 0x3b010000)
the space 12288K, 3% used [0x37010000, 0x3706dac8, 0x3706dc00, 0x37c10000)
```



```
ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

可以看到该对象被分配在了年轻代，占用了 25%的空间。如果需要将 1MB 以上的对象直接在年老代分配，设置-XX:PetenureSizeThreshold=1000000，程序输出如清单 7-37 所示。

代码清单 7-37 示例程序代码运行输出

```
Heap
def new generation total 6144K, used 354K [0x35c10000, 0x362b0000, 0x362b0000)
eden space 5504K, 6% used [0x35c10000, 0x35c689d8, 0x36170000)
from space 640K, 0% used [0x36170000, 0x36170000, 0x36210000)
to space 640K, 0% used [0x36210000, 0x36210000, 0x362b0000)
tenured generation total 13696K, used 1024K [0x362b0000, 0x37010000, 0x37010000)
the space 13696K, 7% used [0x362b0000, 0x363b0010, 0x363b0200, 0x37010000)
compacting perm gen total 12288K, used 374K [0x37010000, 0x37c10000, 0x3b010000)
the space 12288K, 3% used [0x37010000, 0x3706dac8, 0x3706dc00, 0x37c10000)
ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

从上面的输出中，我们可以看到当满 1MB 时进入到了年老代。

7.4.3 设置对象进入年老代的年龄

堆中的每一个对象都有自己的年龄。一般情况下，年轻对象存放在年轻代，年老对象存放在年老代。为了做到这点，虚拟机为每个对象都维护一个年龄。

如果对象在 eden 区，经过一次 GC 后还存活，则被移动到 survivor 区中，对象年龄加 1。以后，对象每经过一次 GC 依然活的，则年龄再加 1。当对象年龄达到阈值时，就移入年老代，成为老年对象。

这个阈值的最大值可以通过参数-XX:MaxTenuringThreshold 来设置，默认值是 15。虽然-XX:MaxTenuringThreshold 的值可能是 15 或者更大，但这不意味着新对象非要达到这个年龄才能进入年老代。事实上，对象实际进入年老代的年龄是虚拟机在运行时根据内存使用情况动态计算的，这个参数指定的是阈值年龄的最大值。即，实际晋升年老代年龄等于动态计算所得的年龄与-XX:MaxTenuringThreshold 中较小的那个。示例代码如代码清单 7-38 所示。

代码清单 7-38 示例程序代码

```
public class MaxTenuringThreshold {
    public static void main(String args[]){
        byte[] b1,b2,b3;
        b1 = new byte[1024*512];
        b2 = new byte[1024*1024*2];
        b3 = new byte[1024*1024*4];
        b3 = null;
        b3 = new byte[1024*1024*4];
    }
}
```

参数设置为: `-XX:+PrintGCDetails -Xmx20M -Xms20M -Xmn10M -XX:SurvivorRatio=2`

运行输出如清单 7-39 所示。

代码清单 7-39 示例程序代码运行输出

```
[GC [DefNew: 2986K->690K(7680K), 0.0246816 secs] 2986K->2738K(17920K), 0.0247226
secs] [Times: user=0.00 sys=0.02, real=0.03 secs]
[GC [DefNew: 4786K->690K(7680K), 0.0016073 secs] 6834K->2738K(17920K), 0.0016436
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
  def new generation   total 7680K, used 4888K [0x35c10000, 0x36610000, 0x36610000)
    eden space 5120K,   82% used [0x35c10000, 0x36029a18, 0x36110000)
    from space 2560K,   26% used [0x36110000, 0x361bc950, 0x36390000)
    to   space 2560K,   0% used [0x36390000, 0x36390000, 0x36610000)
  tenured generation   total 10240K, used 2048K [0x36610000, 0x37010000, 0x37010000)
    the space 10240K,   20% used [0x36610000, 0x36810010, 0x36810200, 0x37010000)
  compacting perm gen  total 12288K, used 374K [0x37010000, 0x37c10000, 0x3b010000)
    the space 12288K,    3% used [0x37010000, 0x3706db50, 0x3706dc00, 0x37c10000)
    ro space 10240K,   51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
    rw space 12288K,   55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

更改参数为: `-XX:+PrintGCDetails -Xmx20M -Xms20M -Xmn10M -XX:SurvivorRatio=2 -XX:MaxTenuringThreshold=1`, 运行输出如清单 7-40 所示。

代码清单 7-40 示例程序代码运行输出

```
[GC [DefNew: 2986K->690K(7680K), 0.0047778 secs] 2986K->2738K(17920K), 0.0048161
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 4888K->0K(7680K), 0.0016271 secs] 6936K->2738K(17920K), 0.0016630
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
  def new generation   total 7680K, used 4198K [0x35c10000, 0x36610000, 0x36610000)
    eden space 5120K,   82% used [0x35c10000, 0x36029a18, 0x36110000)
    from space 2560K,    0% used [0x36110000, 0x36110088, 0x36390000)
    to   space 2560K,    0% used [0x36390000, 0x36390000, 0x36610000)
  tenured generation   total 10240K, used 2738K [0x36610000, 0x37010000, 0x37010000)
    the space 10240K,   26% used [0x36610000, 0x368bc890, 0x368bca00, 0x37010000)
  compacting perm gen  total 12288K, used 374K [0x37010000, 0x37c10000, 0x3b010000)
    the space 12288K,    3% used [0x37010000, 0x3706db50, 0x3706dc00, 0x37c10000)
    ro space 10240K,   51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
    rw space 12288K,   55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

上面加粗的字体显示,第一次运行时 `b1` 对象在程序结束后依然保存在年轻代。第二次运行前,我们减小了对象晋升年老代的年龄,设置为 1。即,所有经过一次 GC 的对象都可以直接进入年老代。程序运行后,可以发现 `b1` 对象已经被分配到年老代。如果希望对象尽可能长时间地停留在年轻代,可以设置一个较大的阈值。

7.4.4 稳定与震荡的堆大小

一般来说，稳定的堆大小是对垃圾回收有利的。获得一个稳定的堆大小的方法是使-Xms 和-Xmx 的大小一致，即最大堆和最小堆（初始堆）一样。如果这样设置，系统在运行时堆大小理论上是恒定的，稳定的堆空间可以减少 GC 的次数。因此，很多服务端应用都会将最大和最小堆设置为相同的数值。

但是，一个不稳定的堆并非毫无用处。稳定的堆大小虽然可以减少 GC 次数，但同时也增加了每次 GC 的时间。让堆大小在一个区间中震荡，在系统不需要使用大内存时，压缩堆空间，使 GC 应对一个较小的堆，可以加快单次 GC 的速度。基于这样的考虑，JVM 还提供了两个参数用于压缩和扩展堆空间。

-XX:MinHeapFreeRatio 参数用来设置堆空间最小空闲比例，默认值是 40。当堆空间的空闲内存小于这个数值时，JVM 便会扩展堆空间。

-XX:MaxHeapFreeRatio 参数用来设置堆空间最大空闲比例，默认值是 70。当堆空间的空闲内存大于这个数值时，便会压缩堆空间，得到一个较小的堆。

当-Xmx 和-Xms 相等时，-XX:MinHeapFreeRatio 和-XX:MaxHeapFreeRatio 两个参数无效。

代码清单 7-41 示例程序代码

```
import java.util.Vector;

public class HeapSize {
    public static void main(String args[]) throws InterruptedException{
        Vector v = new Vector();
        while(true){
            byte[] b = new byte[1024*1024];
            v.add(b);
            if(v.size() == 10){
                v = new Vector();
            }
            Thread.sleep(1);
        }
    }
}
```

上述代码测试-XX:MinHeapFreeRatio 和-XX:MaxHeapFreeRatio 的作用，设置运行参数为-XX:+PrintGCDetails -Xms10M -Xmx40M -XX:MinHeapFreeRatio=40 -XX:MaxHeapFreeRatio=50 时，输出如清单 7-42 所示。

代码清单 7-42 示例程序代码运行输出

```
[GC [DefNew: 2418K->178K(3072K), 0.0034827 secs] 2418K->2226K(9920K), 0.0035249
secs] [Times: user=0.00 sys=0.00, real=0.03 secs]
[GC [DefNew: 2312K->0K(3072K), 0.0028263 secs] 4360K->4274K(9920K), 0.0029905 secs]
[Times: user=0.00 sys=0.00, real=0.03 secs]
[GC [DefNew: 2068K->0K(3072K), 0.0024363 secs] 6342K->6322K(9920K), 0.0024836 secs]
[Times: user=0.00 sys=0.00, real=0.03 secs]
```

```
[GC [DefNew: 2061K->0K(3072K), 0.0017376 secs][Tenured: 8370K->8370K(8904K),
0.1392692 secs] 8384K->8370K(11976K), [Perm : 374K->374K(12288K)], 0.1411363 secs]
[Times: user=0.00 sys=0.02, real=0.16 secs]
[GC [DefNew: 5138K->0K(6336K), 0.0038237 secs] 13508K->13490K(20288K), 0.0038632
secs] [Times: user=0.00 sys=0.00, real=0.03 secs]
```

改用参数: `-XX:+PrintGCDetails -Xms40M -Xmx40M -XX:MinHeapFreeRatio=40 -XX:MaxHeapFreeRatio=50`, 运行输出如清单 7-43 所示。

代码清单 7-43 示例程序代码运行输出

```
[GC [DefNew: 10678K->178K(12288K), 0.0019448 secs] 10678K->178K(39616K), 0.0019851
secs] [Times: user=0.00 sys=0.00, real=0.03 secs]
[GC [DefNew: 10751K->178K(12288K), 0.0010295 secs] 10751K->178K(39616K), 0.0010697
secs] [Times: user=0.00 sys=0.00, real=0.02 secs]
[GC [DefNew: 10493K->178K(12288K), 0.0008301 secs] 10493K->178K(39616K), 0.0008672
secs] [Times: user=0.00 sys=0.00, real=0.02 secs]
[GC [DefNew: 10467K->178K(12288K), 0.0008522 secs] 10467K->178K(39616K), 0.0008905
secs] [Times: user=0.00 sys=0.00, real=0.02 secs]
[GC [DefNew: 10450K->178K(12288K), 0.0008964 secs] 10450K->178K(39616K), 0.0009339
secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
[GC [DefNew: 10439K->178K(12288K), 0.0009876 secs] 10439K->178K(39616K), 0.0010279
secs] [Times: user=0.00 sys=0.00, real=0.02 secs]
```

此时堆空间的垃圾回收稳定在一个固定的范围。在一个稳定的堆中, 堆空间大小始终不变, 每次 GC 时, 都要应对一个 40MB 的空间。因此, 虽然 GC 次数减小了, 但是单次 GC 速度不如一个震荡的堆。

7.4.5 吞吐量优先案例

吞吐量优先的方案将会尽可能减少系统的执行垃圾回收的总时间, 故可以考虑关注系统吞吐量的并行回收收集器。在拥有高性能的计算机上, 进行吞吐量优先优化, 可以使用参数:

```
java -Xmx3800m -Xms3800m -Xmn2G -Xss128k -XX:+UseParallelGC -XX:ParallelGC-Threads=
20 -XX:+UseParallelOldGC
```

-Xmx380m -Xms3800m: 设置 Java 堆的最大值和初始值。一般情况下, 为了避免堆内存的频繁震荡, 导致系统性能下降, 我们的做法是设置最大堆等于最小堆。假设这里把最小堆减少为最大堆的一般, 即 1900m, 那么 JVM 会尽可能在 1900MB 堆空间中运行, 如果这样, 发生 GC 的可能性就会比较高;

-Xss128k: 减少线程栈的大小, 这样可以使剩余的系统内存支持更多的线程;

-Xmn2g: 设置新生代区域大小为 2GB;

-XX:+UseParallelGC: 新生代使用并行垃圾回收收集器。这是一个关注吞吐量的收集器, 可以尽可能地减少 GC 时间。

-XX:ParallelGC-Threads: 设置用于垃圾回收的线程数, 通常情况下, 可以设置和 CPU 数量相等。但在 CPU 数量比较多的情况下, 设置相对较小的数值也是合理的;

- XX:+UseParallelOldGC: 设置老年代使用并行回收收集器。

7.4.6 使用大页案例

在 Solaris 系统中, JVM 可以支持大页的使用。使用大的内存分页可以增强 CPU 的内存寻址能力, 从而提升系统的性能。

```
java -Xmx2506m -Xms2506m -Xmn1536m -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20 -XX:+UseParallelOldGC -XX:+LargePageSizeInBytes=256m
```

- XX:+LargePageSizeInBytes: 设置大页的大小。

7.4.7 降低停顿案例

为降低应用软件的垃圾回收时的停顿, 首先考虑的是使用关注系统停顿的 CMS 回收器, 其次, 为了减少 Full GC 次数, 应尽可能将对象预留在新生代, 因为新生代 Minor GC 的成本远远小于老年代的 Full GC。

```
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+SurvivorRatio=8 -XX:TargetSurvivorRatio=90 -XX:MaxTenuringThreshold=31
```

- XX:ParallelGCThreads=20: 设置 20 个线程进行垃圾回收;

- XX:+UseParNewGC: 新生代使用并行回收器;

- XX:+UseConcMarkSweepGC: 老年代使用 CMS 收集器降低停顿;

- XX:+SurvivorRatio: 设置 eden 区和 survivor 区的比例为 8:1。稍大的 survivor 空间可以提高在新生代回收生命周期较短的对象的可能性, 如果 survivor 不够大, 一些短命的对象可能直接进入老年代, 这对系统来说是不利的。

- XX:TargetSurvivorRatio=90: 设置 survivor 区的可使用率。这里设置为 90%, 则允许 90% 的 survivor 空间被使用。默认值是 50%。故该设置提高了 survivor 区的使用率。当存放的对象超过这个百分比, 则对象会向老年代压缩。因此, 这个选项更有助于将对象留在新生代。

- XX:MaxTenuringThreshold: 设置年轻对象晋升到老年代的年龄。默认值是 15 次, 即对象经过 15 次 Minor GC 依然存活, 则进入老年代。这里设置为 31, 目的是让对象尽可能地保存在新生代区域。

7.4.8 设置最大堆内存

JVM 内存结构分配对 Java 应用程序的性能有较大的影响。

Java 应用程序可以使用的最大堆可以用 -Xmx 参数指定。最大堆指的是年轻代和年老代的大小之和的最大值, 它是 Java 应用程序的堆上限。以下代码演示在堆上分配空间直到内存溢出。-Xmx 参数的大小不同, 将直接决定程序能够走过几个循环。

代码清单 7-44 示例程序代码

```
import java.util.Vector;

public class maxHeapTest {
```

```
public static void main(String[] args){
    Vector v = new Vector();
    for(int i=0;i<=10;i++){
        byte[] b = new byte[1024*1024];
        v.add(b);
        System.out.println(i+"M is allocated");
    }
    System.out.println("Max memory:"+Runtime.getRuntime().maxMemory());
}
```

使用-Xmx5M 设置时最大堆上限为 5MB，此时系统输出如清单 7-45 所示。

代码清单 7-45 示例程序代码运行输出

```
0M is allocated
1M is allocated
2M is allocated
3M is allocated
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at maxHeapTest.main(maxHeapTest.java:8)
```

此时表明在完成 4MB 数据分配后系统空闲的堆内存大小已经不足 1MB 了。

如果设置最大堆上限为 11MB，此时程序顺利结束，没有任何异常。表明在 11MB 的堆空间上成功分配了 10MB 的字节数组。

代码清单 7-46 示例程序代码运行输出

```
0M is allocated
1M is allocated
2M is allocated
3M is allocated
4M is allocated
5M is allocated
6M is allocated
7M is allocated
8M is allocated
9M is allocated
10M is allocated
Max memory:16252928
```

7.4.9 设置最小堆内存

使用 JVM 参数-Xms 可以用于设置最小堆内存空间，也就是 JVM 启动时所占据的操作系统内存大小。

Java 应用程序在运行时首先会被分配-Xms 指定的内存大小，并尽可能尝试在这个空间段内运行程序。当-Xms 指定的内存大小确实无法满足应用程序时，JVM 才会向操作系统申请更多的内存，直到内存大小达到-Xmx 指定的最大内存为止。若超过-Xmx 的值，则抛出 OutOfMemoryError 异常。

如果-Xms 的数值较小, 那么 JVM 为了保证系统尽可能地在指定内存范围内运行就会更加频繁地进行 GC 操作, 以释放失效的内存空间, 从而会增加 Minor GC 和 Full GC 的次数, 对系统性能产生一定的影响。

代码清单 7-47 所示的代码每次分配 1MB 空间, 累计 3MB 时清空内存。

代码清单 7-47 示例程序代码

```
import java.util.Vector;

public class minHeapTest {
    public static void main(String[] args){
        Vector v = new Vector();
        for(int i=1;i<=10;i++){
            byte[] b = new byte[1024*1024];
            v.add(b);
            if(v.size()==3){
                v.clear();
            }
        }
    }
}
```

当设置为-XX:+PrintGCDetails -Xmx11M -Xms1M -verbose:gc 时抛出错误:

```
Error occurred during initialization of VM
Too small initial heap for new size specified
```

如果设置最小堆内存为 1.5MB, 则 GC 输出:

代码清单 7-48 示例程序代码运行输出

```
[GC [DefNew: 227K->64K(960K), 0.0018943 secs][Tenured: 85K->149K(1024K), 0.0097617
secs] 227K->149K(1984K), [Perm : 375K->375K(12288K)], 0.0117270 secs] [Times: user=0.02
sys=0.00, real=0.02 secs]

[GC [DefNew: 18K->0K(960K), 0.0003687 secs][Tenured: 1173K->1173K(2052K),
0.0079005 secs] 1191K->1173K(3012K), [Perm : 375K->375K(12288K)], 0.0083541 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]

[GC [DefNew: 0K->0K(1024K), 0.0001338 secs][Tenured: 2197K->2197K(3080K),
0.0080497 secs] 2197K->2197K(4104K), [Perm : 375K->375K(12288K)], 0.0082739 secs]
[Times: user=0.01 sys=0.00, real=0.02 secs]
clearing....

[GC [DefNew: 1056K->0K(1728K), 0.0001662 secs] 3253K->2197K(5392K), 0.0001989 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]

[GC [DefNew: 1024K->0K(1728K), 0.0009438 secs] 3221K->3221K(5392K), 0.0009781 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]

[GC [DefNew (promotion failed) : 1024K->1024K(1728K), 0.0001701 secs][Tenured:
3221K->2197K(4096K), 0.0085937 secs] 4245K->2197K(5824K), [Perm : 375K->375K(12288K)],
0.0088455 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
clearing....

[GC [DefNew: 1024K->0K(1728K), 0.0001476 secs] 3221K->2197K(5392K), 0.0001749 secs]
```

```
[Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 1024K->0K(1728K), 0.0003679 secs] 3221K->3221K(5392K), 0.0003979 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 1024K->1024K(1728K), 0.0000162 secs][Tenured: 3221K->2197K(3664K),
0.0085309 secs] 4245K->2197K(5392K), [Perm : 375K->375K(12288K)], 0.0086110 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
clearing....
[GC [DefNew: 1024K->0K(1728K), 0.0001591 secs] 3221K->2197K(5392K), 0.0001875 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
def new generation    total 1728K, used 1056K [0x36a10000, 0x36be0000, 0x36c10000)
eden space 1600K,    66% used [0x36a10000, 0x36b18080, 0x36ba0000)
from space 128K,     0% used [0x36ba0000, 0x36ba0000, 0x36bc0000)
to   space 128K,     0% used [0x36bc0000, 0x36bc0000, 0x36be0000)
tenured generation    total 3664K, used 2197K [0x36c10000, 0x36fa4000, 0x37010000)
the space 3664K,     59% used [0x36c10000, 0x36e354b8, 0x36e35600, 0x36fa4000)
compacting perm gen    total 12288K, used 375K [0x37010000, 0x37c10000, 0x3b010000)
the space 12288K,     3% used [0x37010000, 0x3706dc38, 0x3706de00, 0x37c10000)
ro space 10240K,      51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
rw space 12288K,      55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

其中进行了 10 次 Minor GC，并共计耗时 4.4 毫秒。为减少 GC 次数，增大-Xms 的值，使用 11MB，输出如下。

代码清单 7-49 示例程序代码

```
[GC [DefNew: 2310K->149K(3392K), 0.0033722 secs] 2310K->2197K(10944K), 0.0034132
secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
clearing....
[GC [DefNew: 2231K->149K(3392K), 0.0016030 secs] 4279K->3221K(10944K), 0.0016354
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
clearing....
[GC [DefNew: 2281K->149K(3392K), 0.0008025 secs] 5353K->3221K(10944K), 0.0008337
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 2211K->149K(3392K), 0.0024706 secs] 5284K->5269K(10944K), 0.0025057
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
clearing....
Heap
def new generation    total 3392K, used 2268K [0x36410000, 0x367b0000, 0x36810000)
eden space 3072K,    68% used [0x36410000, 0x36621dd0, 0x36710000)
from space 320K,     46% used [0x36710000, 0x36735498, 0x36760000)
to   space 320K,     0% used [0x36760000, 0x36760000, 0x367b0000)
tenured generation    total 7552K, used 5120K [0x36810000, 0x36f70000, 0x37010000)
the space 7552K,     67% used [0x36810000, 0x36d10050, 0x36d10200, 0x36f70000)
compacting perm gen    total 12288K, used 375K [0x37010000, 0x37c10000, 0x3b010000)
the space 12288K,     3% used [0x37010000, 0x3706dc88, 0x3706de00, 0x37c10000)
ro space 10240K,      51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
rw space 12288K,      55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```


增大-Xms 后，只进行了 4 次 Minor GC。

JVM 会试图将系统内存尽可能限制在-Xms 中，因此当内存实际使用量触及-Xms 指定的大小时会触发 Full GC。因此把-Xms 值设置为-Xmx 时，可以在系统运行初期减少 GC 的次数和耗时。

7.4.10 设置年轻代

参数-Xmn 或者用于 Hot Spot 虚拟机中的参数-XX:NewSize(年轻代初始大小)、-XX:MaxNewSize 用于设置年轻代的大小。设置一个较大的年轻代会减小年老代的大小，这个参数对系统性能以及 GC 行为有很大的影响。年轻代的大小一般设置为整个堆空间的 1/4 到 1/3 左右。

以上例中的代码为例，若使用 JVM 参数-XX:+PrintGCDetails -Xmx11M -XX:NewSize=2M -XX:MaxNewSize=2M -verbose:gc 运行程序，将年轻代的大小减小为 2MB，那么 MinorGC 次数将从 4 次增加到 9 次(默认情况下是 3.5MB 左右)。运行输出如清单 7-50 所示。

代码清单 7-50 示例程序代码运行输出

```
[GC [DefNew: 1272K->150K(1856K), 0.0028101 secs] 1272K->1174K(11072K), 0.0028504
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 1174K->0K(1856K), 0.0018805 secs] 2198K->2198K(11072K), 0.0019097
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
clearing....
[GC [DefNew: 1076K->0K(1856K), 0.0004046 secs] 3274K->2198K(11072K), 0.0004382
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 1024K->0K(1856K), 0.0011834 secs] 3222K->3222K(11072K), 0.0013508
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 1024K->0K(1856K), 0.0012983 secs] 4246K->4246K(11072K), 0.0013299
secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
clearing....
[GC [DefNew: 1024K->0K(1856K), 0.0001441 secs] 5270K->4246K(11072K), 0.0001686
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 1024K->0K(1856K), 0.0012028 secs] 5270K->5270K(11072K), 0.0012328
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 1024K->0K(1856K), 0.0012553 secs] 6294K->6294K(11072K), 0.0012845
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
clearing....
[GC [DefNew: 1024K->0K(1856K), 0.0001524 secs] 7318K->6294K(11072K), 0.0001780
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
def new generation    total 1856K, used 1057K [0x36410000, 0x36610000, 0x36610000)
  eden space 1664K,    63% used [0x36410000, 0x365185a0, 0x365b0000)
  from space 192K,     0% used [0x365e0000, 0x365e0088, 0x36610000)
  to   space 192K,     0% used [0x365b0000, 0x365b0000, 0x365e0000)
tenured generation    total 9216K, used 6294K [0x36610000, 0x36f10000, 0x37010000)
  the space 9216K,    68% used [0x36610000, 0x36c35868, 0x36c35a00, 0x36f10000)
compacting perm gen   total 12288K, used 375K [0x37010000, 0x37c10000, 0x3b010000)
  the space 12288K,    3% used [0x37010000, 0x3706dc88, 0x3706de00, 0x37c10000)
   ro space 10240K,   51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
   rw space 12288K,   55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

7.4.11 设置持久代

持久代（方法区）不属于堆的一部分。在 Hot Spot 虚拟机中，使用-XX:MaxPermSize 参数可以设置持久代的最大值，使用-XX:PermSize 可以设置持久代的初始大小。

持久代的大小直接决定了系统可以支持多少个类定义和多少常量。对于使用 CGLIB 或者 Javassist 等动态字节码生成工具的应用程序而言，设置合理的持久代大小有助于维持系统稳定。

系统所支持的最大类与 MaxPermSize 成正比。一般来说，MaxPermSize 设置为 64MB 已经可以满足绝大部分应用程序正常工作。如果依然出现永久区溢出，可以设置为 128MB。这是两个很常用的永久区取值。如果 128MB 依然不能满足应用程序需求，那么对于大部分应用程序来说，则应该考虑优化系统的设计，减少动态类的产生，或者利用 GC 回收部分驻扎在永久区的无用类信息，以使系统健康运行。

7.4.12 设置线程栈

线程栈是线程的一块私有空间。在 JVM 中可以使用-Xss 参数设置线程栈的大小。

在线程中进行局部变量分配，函数调用时都需要在栈中开辟空间。如果栈的空间分配太小，那么线程在运行时可能没有足够的空间分配局部变量或者达不到足够的函数调用深度，导致程序异常退出；如果栈空间过大，那么开设线程所需的内存成本就会上升，系统所能支持的线程总数就会下降。

由于 Java 堆也是向操作系统申请内存空间的，因此，如果堆空间过大，就会导致操作系统用于线程栈的内存减少，从而间接减少程序所能支持的线程数量。

代码清单 7-51 所示代码尝试开设尽可能多的线程，并在线程数量饱和时，打印已经开设的线程数量。

代码清单 7-51 示例程序代码

```
public class TestXss {
    public static class MyThread extends Thread{
        @Override
        public void run(){
            try{
                Thread.sleep(10000);
            }catch(InterruptedException e){
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args){
        int count=0;
        try{
            for(int i=0;i<10000;i++){
                new MyThread().start();
            }
        }
    }
}
```



```

        count++;
    }
} catch (OutOfMemoryError e) {
    System.out.println(count);
    System.out.println(e.getMessage());
}
}
}

```

设置-Xss1M 时运行这个程序，即设置每个线程拥有 1MB 的栈空间。输出如下：

```

1578
unable to create new native thread

```

一共允许启动 1578 个线程。

如果增大-Xss 的值，使用参数-Xss20M，为每个线程分配 20M 的栈空间，结果显示如下：

```

69
unable to create new native thread

```

如果改变系统的最大堆空间设定，可以发现系统所能支持的线程数量也会相应改变。

```

-Xmx100M -Xms10M -Xss1M
1728
unable to create new native thread

```

```

-Xmx100M -Xms10M -Xss1M
844
unable to create new native thread

```

```

-Xmx1000M -Xms1000M -Xss10M
74
unable to create new native thread

```

Java 堆的分配以 200MB 递增，当栈大小为 1MB 时，最大线程数量以 200 递减。当系统物理内存被堆占据时，就不可以被栈使用。

当系统由于内存空间不够而无法创建新的线程时会抛出 OOM 异常。这并不是由于堆内存不够而导致的 OOM，而是因为操作系统内存减去堆内存后剩余的系统内存不足而无法创建新的线程。在这种情况下可以尝试减少堆内存以换取更多的系统空间来解决这个问题。

综上所述，如果系统确实需要大量线程并发执行，那么设置一个较小的堆和较小的栈有助于提高系统所能承受的最大线程数。

7.4.13 堆的比例分配

参数-XX: SurvivorRatio 是用来设置年轻代中 eden 空间和 s0 空间的比例关系。s0 和 s1 空间又分别称为 from 空间和 to 空间。它们的大小是相同的，职能也是相同的，并在 Minor GC 后互换角色。

代码清单 7-50 所示的例子演示不断插入字符时使用设置年轻代堆为 10MB，并使 eden 区是 s0

的 8 倍大小,

```
-XX:+PrintGCDetails -XX:MaxNewSize=10M -XX:SurvivorRatio=8
```

代码清单 7-52 示例程序代码

```
import java.util.ArrayList;
import java.util.List;

public class StringDemo {
    public static void main(String[] args){
        List<String> handler = new ArrayList<String>();
        for(int i=0;i<1000;i++){
            HugeStr h = new HugeStr();
            ImprovedHugeStr hl = new ImprovedHugeStr();
            handler.add(h.getSubString(1, 5));
            handler.add(hl.getSubString(1, 5));
        }
    }

    static class HugeStr{
        private String str = new String(new char[800000]);
        public String getSubString(int begin,int end){
            return str.substring(begin, end);
        }
    }

    static class ImprovedHugeStr{
        private String str = new String(new char[10000000]);
        public String getSubString(int begin,int end){
            return new String(str.substring(begin, end));
        }
    }
}
```

GC 输出如清单 7-53 所示。

代码清单 7-53 示例程序代码运行输出

```
[Full GC [Tenured: 233756K->233743K(251904K), 0.0524229 secs] 233756K->233743K
(261120K), [Perm : 377K->372K(12288K)], 0.0524703 secs] [Times: user=0.06 sys=0.00,
real=0.06 secs]
def new generation    total 9216K, used 170K [0x27010000, 0x27a10000, 0x27a10000)
  eden space 8192K,    2% used [0x27010000, 0x2703a978, 0x27810000)
  from space 1024K,    0% used [0x27910000, 0x27910000, 0x27a10000)
  to   space 1024K,    0% used [0x27810000, 0x27810000, 0x27910000)
tenured generation    total 251904K, used 233743K [0x27a10000, 0x37010000,
0x37010000)
  the space 251904K,   92% used [0x27a10000, 0x35e53d00, 0x35e53e00, 0x37010000)
compacting perm gen    total 12288K, used 372K [0x37010000, 0x37c10000, 0x3b010000)
```



```

the space 12288K, 3% used [0x37010000, 0x3706d310, 0x3706d400, 0x37c10000)
ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)

```

修改参数, SurvivorRatio 改为 2 运行程序, 相当于设置 eden 区是 s0 的 2 倍大小, 由于 s1 与 s0 相同, 故有 $\text{eden}=[10\text{MB}/(1+1+2)]*2=5\text{MB}$ 。输出如清单 7-54 所示。

代码清单 7-54 示例程序代码

```

[Full GC [Tenured: 233756K->233743K(251904K), 0.0546689 secs]
233756K->233743K(259584K), [Perm : 377K->372K(12288K)], 0.0547257 secs] [Times:
user=0.05 sys=0.00, real=0.05 secs]

def new generation total 7680K, used 108K [0x27010000, 0x27a10000, 0x27a10000)
eden space 5120K, 2% used [0x27010000, 0x2702b3b0, 0x27510000)
from space 2560K, 0% used [0x27510000, 0x27510000, 0x27790000)
to space 2560K, 0% used [0x27790000, 0x27790000, 0x27a10000)
tenured generation total 251904K, used 233743K [0x27a10000, 0x37010000,
0x37010000)
the space 251904K, 92% used [0x27a10000, 0x35e53d00, 0x35e53e00, 0x37010000)
compacting perm gen total 12288K, used 372K [0x37010000, 0x37c10000, 0x3b010000)
the space 12288K, 3% used [0x37010000, 0x3706d310, 0x3706d400, 0x37c10000)
ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)

```

参数-XX: NewRatio 可以用来设置年轻代和年老代的比例:

-XX: NewRatio=年老代/年轻代

使用参数-XX:+PrintGCDetails -XX:NewRatio=2 -Xmx20M -Xms20M 运行上述代码, 输出如清单 7-55 所示。

代码清单 7-55 示例程序代码运行输出

```

[GC [DefNew: 3369K->150K(6144K), 0.0030730 secs][Tenured: 1562K->1712K(13696K),
0.0098075 secs] 3369K->1712K(19840K), [Perm : 377K->377K(12288K)], 0.0129554 secs]
[Times: user=0.02 sys=0.00, real=0.01 secs]

[Full GC [Tenured: 1712K->1699K(13696K), 0.0081938 secs] 1712K->1699K(19840K),
[Perm : 377K->372K(12288K)], 0.0082420 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]

Heap
def new generation total 6144K, used 168K [0x35c10000, 0x362b0000, 0x362b0000)
eden space 5504K, 3% used [0x35c10000, 0x35c3a358, 0x36170000)
from space 640K, 0% used [0x36210000, 0x36210000, 0x362b0000)
to space 640K, 0% used [0x36170000, 0x36170000, 0x36210000)
tenured generation total 13696K, used 1699K [0x362b0000, 0x37010000, 0x37010000)
the space 13696K, 12% used [0x362b0000, 0x36458dd8, 0x36458e00, 0x37010000)
compacting perm gen total 12288K, used 372K [0x37010000, 0x37c10000, 0x3b010000)
the space 12288K, 3% used [0x37010000, 0x3706d218, 0x3706d400, 0x37c10000)
ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)

```

在本例中, 因为堆大小为 20MB, 年轻代和年老代的比为 1: 2, 故年轻代大小大约为

20MB*1/3=6MB，年老代为 12MB 左右。

如果改为-XX:+PrintGCDetails -XX:NewRatio=1 -Xmx20M -Xms20M，输出如清单 7-56 所示。

代码清单 7-56 示例程序代码运行输出

```
[GC [DefNew: 3468K->150K(9216K), 0.0028638 secs][Tenured: 1562K->1712K(10240K),
0.0084220 secs] 3468K->1712K(19456K), [Perm : 377K->377K(12288K)], 0.0113816 secs]
[Times: user=0.02 sys=0.00, real=0.01 secs]
[Full GC [Tenured: 1712K->1699K(10240K), 0.0071963 secs] 1712K->1699K(19456K),
[Perm : 377K->372K(12288K)], 0.0072393 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
Heap
def new generation      total 9216K, used 491K [0x35c10000, 0x36610000, 0x36610000)
  eden space 8192K,      6% used [0x35c10000, 0x35c8af08, 0x36410000)
  from space 1024K,      0% used [0x36510000, 0x36510000, 0x36610000)
  to   space 1024K,      0% used [0x36410000, 0x36410000, 0x36510000)
tenured generation      total 10240K, used 1699K [0x36610000, 0x37010000, 0x37010000)
  the space 10240K,      16% used [0x36610000, 0x367b8dd8, 0x367b8e00, 0x37010000)
compacting perm gen      total 12288K, used 372K [0x37010000, 0x37c10000, 0x3b010000)
  the space 12288K,      3% used [0x37010000, 0x3706d218, 0x3706d400, 0x37c10000)
  ro space 10240K,      51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
  rw space 12288K,      55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

虽然没有完全成为 1: 1(实际为 3:4 左右)，但是已经趋近于 1:1。

7.4.14 堆分配参数总结

JVM 内存区域划分如图 7-3 所示。

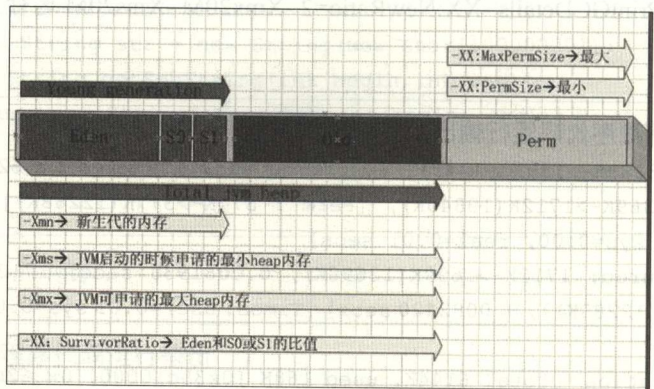


图7-3 JVM内存区域划分图

与 Java 应用程序堆内存相关的 JVM 参数有：

-Xmn: 新生代内存大小的最大值，包括 E 区和两个 S 区的总和，使用方法如：-Xmn65535, -Xmn1024k, -Xmn512m, -Xmn1g。-Xmn 只能使用在 JDK1.4 或之后的版本中，（之前的 1.3/1.4 版本中，可使用-XX:NewSize 设置年轻代大小，用-XX:MaxNewSize 设置年轻代最大值）；如果同时设置了-Xmn 和-XX:NewSize, -XX:MaxNewSize, 则谁设置在后面，谁就生效。如果同时设置了-XX:NewSize -XX:MaxNewSize 与-XX:NewRatio 则实际生效的值是 min(MaxNewSize,max (NewSize, heap/(NewRatio+1)))。在开发、测试环境，可以-XX:NewSize 和 -XX:MaxNewSize 来设

置新生代大小，但在线上生产环境，使用 `-Xmn` 一个即可（推荐），或者将 `-XX:NewSize` 和 `-XX:MaxNewSize` 设置为同一个值，这样能够防止在每次 GC 之后都要调整堆的大小（即：抖动，抖动会严重影响性能）。

`-Xms`：设置 Java 应用程序启动时的初始堆大小，也是堆大小的最小值，默认值是总共的物理内存/64（且小于 1G），默认情况下，当堆中可用内存小于 40%（这个值可以用 `-XX:MinHeapFreeRatio` 调整，如 `-X:MinHeapFreeRatio=30`）时，堆内存会开始增加，一直增加到 `-Xmx` 的大小；

`-Xmx`：设置 Java 应用程序能获得的最大堆大小，默认值是总共的物理内存/64（且小于 1G），如果 `Xms` 和 `Xmx` 都不设置，则两者大小会相同，默认情况下，当堆中可用内存大于 70%（这个值可以用 `-XX:MaxHeapFreeRatio` 调整，如 `-X:MaxHeapFreeRatio=60`）时，堆内存会开始减少，一直减小到 `-Xms` 的大小。整个堆的大小=年轻代大小+年老代大小，堆的大小不包含持久代大小，如果增大了年轻代，年老代相应就会减小，官方默认的配置为年老代大小/年轻代大小=2/1 左右（使用 `-XX:NewRatio` 可以设置 `-XX:NewRatio=5`，表示年老代/年轻代=5/1）。建议在开发测试环境可以用 `Xms` 和 `Xmx` 分别设置最小值最大值，但是在线上生产环境，`Xms` 和 `Xmx` 设置的值必须一样，原因与年轻代一样——防止抖动；

`-Xss`：设置线程栈的大小，默认 1M，一般来说是不需要改的。除非代码不多，可以设置的小点，另外一个相似的参数是 `-XX:ThreadStackSize`，这两个参数在 1.6 以前，都是谁设置在后面，谁就生效；1.6 版本以后，`-Xss` 设置在后面，则以 `-Xss` 为准，`-XXThreadStackSize` 设置在后面，则主线程以 `-Xss` 为准，其他线程以 `-XX:ThreadStackSize` 为准；

`-XX:MinHeapFreeRatio`：设置堆空间最小空闲比例。当堆空间的空闲内存小于这个数值时，JVM 便会扩展堆空间；

`-XX:MaxHeapFreeRatio`：设置堆空间的最大空闲比例。当堆空间的空闲内存大于这个数值时，便会压缩堆空间，得到一个较小的堆；

`-XX:NewSize`：设置年轻代的大小；

`-XX:NewRatio`：设置年老代与年轻代的比例，它等于年老代大小除以年轻代大小；

`-XX:SurvivorRatio`：年轻代中 eden 区与 survivor 区的比例；

`-XX:MaxPermSize`：设置最大的持久区大小；

`-XX:TargetSurvivorRatio`：设置 survivor 区的可使用率。当 survivor 区的空间使用率达到这个数值时，会将对象送入年老代。

7.4.15 垃圾回收器相关参数总结

与串行回收器相关的参数

`-XX:+UseSerialGC`：在年轻代和年老代使用串行回收器。

`-XX:+SuivivorRatio`：设置 eden 区大小和 survivor 区大小的比例。

`-XX:+PretenureSizeThreshold`：设置大对象直接进入年老代的阈值。当对象的大小超过这个值时，将直接在年老代分配。

`-XX:MaxTenuringThreshold`：设置对象进入年老代的年龄的最大值。每一次 Minor GC 后，对

象年龄就加 1。任何大于这个年龄的对象，一定会进入年老代。

与并行 GC 相关的参数

-XX:+UseParNewGC: 在年轻代使用并行收集器。

-XX:+UseParallelOldGC: 年老代使用并行回收收集器。

-XX:ParallelGCThreads: 设置用于垃圾回收的线程数。通过情况下可以和 CPU 数量相等。但在 CPU 数量较多的情况下，设置相对较小的数值也是合理的。

-XX:MaxGCPauseMills: 设置最大垃圾收集停顿时间。它的值是一个大于 0 的整数。收集器在工作时，会调整 Java 堆大小或者其他一些参数，尽可能地把停顿时间控制在 MaxGCPauseMills 以内。

-XX:GCTimeRatio: 设置吞吐量大小，它的值是一个 0-100 之间的整数。假设 GCTimeRatio 的值为 n，那么系统将花费不超过 $1/(1+n)$ 的时间用于垃圾收集。

-XX:+UseAdaptiveSizePolicy: 打开自适应 GC 策略。在这种模式下，年轻代的大小，eden 和 survivor 的比例、晋升年老代的对象年龄等参数会被自动调整，已达到在堆大小、吞吐量和停顿时间之间的平衡点。

与 CMS 回收器相关的参数

-XX:+UseConcMarkSweepGC: 年轻代使用并行收集器，年老代使用 CMS+串行收集器。

-XX:+ParallelCMSThreads: 设定 CMS 的线程数量。

-XX:+CMSInitiatingOccupancyFraction: 设置 CMS 收集器在年老代空间被使用多少后触发，默认为 60%。

-XX:+UseFullGCsBeforeCompaction: 设定进行多少次 CMS 垃圾回收后，进行一次内存压缩。

-XX:+CMSClassUnloadingEnabled: 允许对类元数据进行回收。

-XX:+CMSParallelRemarkEndable: 启用并行重标记。

-XX:CMSInitatingPermOccupancyFraction: 当永久区占用率达到这一百分比后，启动 CMS 回收（前提是-XX:+CMSClassUnloadingEnabled 激活了）。

-XX:UseCMSInitatingOccupancyOnly: 表示只在到达阈值的时候，才进行 CMS 回收。

-XX:+CMSIncrementalMode: 使用增量模式，比较适合单 CPU。

与 G1 回收器相关的参数

-XX:+UseG1GC, 使用 G1 回收器。

-XX:+UnlockExperimentalVMOptions: 允许使用实验性参数。

-XX:+MaxGCPauseMills: 设置最大垃圾收集停顿时间。

-XX:+GCPauseIntervalMills: 设置停顿间隔时间。

其他参数

-XX:+DisableExplicitGC: 禁用显示 GC。

JIT 编译参数

JVM 的 Just-in-time 编译器，可以在运行时将字节码编译成本地代码，从而提高函数的执行效率。-XX:CompileThreshold 为 JIT 编译的阈值，当函数的调用次数超过-XX:CompileThreshold 时，JIT 就将字节码编译成本地机器码。在 Client 模式下，-XX:CompileThreshold 的取值是 1500；在 Server 模式下，取值是 10000。JIT 编译完成后，JVM 便会用本地代码代替原来的字节码解释执行，因此，在系统的未来运行中，这些时间是可以被赚回来的。

JIT 编译会花费一定的时间，为了能合理地设置 JIT 编译的阈值，可以使用-XX:+CITime 打印出 JIT 编译的耗时，也可以使用-XX:+PrintCompilation 打印出 JIT 编译的信息。

采用参数-XX:CompileThreshold=1500 -XX:+PrintCompilation -XX:+CITime 运行代码清单 7-57 所示的这个例子。

代码清单 7-57 示例程序代码

```
public class JITDemoTest {
    static long loopCount = 0;

    //测试 JIT 编译函数
    public static void useJIT(){
        loopCount++;
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    public static void main(String args[]){
        long start = System.currentTimeMillis();
        for(int i=0;i<1496;i++){
            JITDemoTest.useJIT();//循环调用 useJIT 方法，让 loopCount 自增，Client 小于
1500 次不会被 JIT 编译
        }
        System.out.println(System.currentTimeMillis() - start);
    }
}
```

这里只会循环 1496 次，小于设定的 1500 次，所以不会被 JIT 编译成本地代码 useJIT 方法，增加参数-XX:+PrintCompilation 打印出来的，输出如清单 7-58 所示。

代码清单 7-58 示例程序代码运行输出

```
138 1 java.lang.String::hashCode (55 bytes)
```

```
139 2 java.lang.String::equals (81 bytes)
142 3 java.lang.String::lastIndexOf (52 bytes)
142 4 java.lang.String::charAt (29 bytes)
144 5 java.lang.String::indexOf (70 bytes)
147 6 java.lang.String::indexOf (166 bytes)
153 7 java.lang.AbstractStringBuilder::ensureCapacityInternal (16 bytes)
1590
(-XX:+CTime 打印出来的)
Accumulated compiler times (for compiled methods only)
-----
Total compilation time : 0.003 s
Standard compilation : 0.003 s, Average : 0.000
On stack replacement : 0.000 s, Average : -1.#IO
Detailed CI Timings
Setup time: 0.000 s ( 0.0%)
Build IR: 0.001 s (44.3%)
Optimize: 0.000 s ( 5.5%)
Emit LIR: 0.001 s (37.7%)
LIR Gen: 0.000 s ( 9.7%)
Linear Scan: 0.001 s (27.1%)
LIR Schedule: 0.000 s ( 0.0%)
Code Emission: 0.000 s (11.0%)
Code Installation: 0.000 s ( 6.9%)
Instruction Nodes: 344 nodes

Total compiled bytecodes : 480 bytes
Standard compilation : 480 bytes
On stack replacement : 0 bytes
Average compilation speed: 137479 bytes/s

nmethod code size : 1984 bytes
nmethod total size : 4476 bytes
```

如果把循环次数提高到 1501 次，程序运行输出如清单 7-59 所示。

代码清单 7-59 示例程序代码运行输出

```
140 1 java.lang.String::hashCode (55 bytes)
142 2 java.lang.String::equals (81 bytes)
144 3 java.lang.String::lastIndexOf (52 bytes)
144 4 java.lang.String::charAt (29 bytes)
145 5 java.lang.String::indexOf (70 bytes)
148 6 java.lang.String::indexOf (166 bytes)
151 7 java.lang.AbstractStringBuilder::ensureCapacityInternal (16 bytes)
1720 8 ! JITDemoTest::useJIT (21 bytes)
1725 9 n java.lang.Thread::sleep (native) (static)
1605
```

Accumulated compiler times (for compiled methods only)


```

-----
Total compilation time   : 0.005 s
Standard compilation     : 0.005 s, Average : 0.001
On stack replacement    : 0.000 s, Average : -1.#IO
Detailed C1 Timings
Setup time:              0.000 s ( 0.0%)
Build IR:                0.002 s (45.0%)
Optimize:                0.000 s ( 4.7%)
Emit LIR:                0.001 s (37.6%)
LIR Gen:                 0.000 s (10.1%)
Linear Scan:             0.001 s (26.6%)
LIR Schedule:            0.000 s ( 0.0%)
Code Emission:           0.000 s (11.4%)
Code Installation:       0.000 s ( 5.9%)
Instruction Nodes:       366 nodes

Total compiled bytecodes : 501 bytes
Standard compilation     : 501 bytes
On stack replacement    : 0 bytes
Average compilation speed: 104400 bytes/s

nmethod code size       : 2176 bytes
nmethod total size      : 4832 bytes

```

通过增大-XX:CompileThreshold 的值, 经过 JIT 编译的函数个数便会减小, 同时, 在 JIT 上消耗的时间也会降低。但对于长期运行的系统, 其性能未必能得到改进, JIT 编译成本地方法是一种加快本地程序运行速度的极好的方法。

堆快照 (Heap Dump)

使用-XX:+HeapDumpOnOutOfMemoryError 参数在程序发生 OOM 时, 导出应用程序的当前堆快照。当程序发生 OOM 而推出系统时, 一些瞬时信息都随着程序的终止而消失, 重现 OOM 问题往往比较困难或者耗时。通过-XX:+HeapDumpPath 可以指定堆快照的保存位置。

```
-Xmx10M -XX:+HeapDumpOnOutOfMemoryError -XX:+HeapDumpPath=C:\my.hprof
```

导出的 Dump 文件可以通过 Visual VM 等工具查看分析, 进而定位问题原因。

当系统发生 OOM 错误时, 可以让虚拟机在错误时运行一段第三方脚本。比如, 当 OOM 发生时, 重置系统: -XX:OnOutOfMemoryError=C:\reset.bat

如果需要在 GC 发生的时刻打印 GC 发生的时间, 则可以追加使用-XX:+PrintGCTimeStamps 选项。打开这个开关后, 将额外输出 GC 的发生时间, 以此可以知道 GC 的频率和间隔。

如果需要查看新生对象晋升老年代的实际阈值, 可以使用参数-XX:+PrintTenuringDistribution 查看。

如果需要在 GC 时打印详细的堆信息, 可以打开-XX:+PrintHeapAtGC 开关。一旦打开它, 那么每次 GC 时, 都将打印堆的使用情况, 这个输出量将是巨大的。

如果需要查看 GC 与应用程序相互执行的耗时, 可以使用-XX:+PrintGCApplicationStoppedTime

和-XX:+PrintGCApplicationConcurrentTime 参数。它们将分别显示应用程序在 GC 发生时的停顿时间和应用程序在 GC 停顿时间的执行时间。可以使用-Xloggc 参数指定 GC 日志的输出位置，如：-Xloggc:C:\gc.log。

JVM 还提供了一组参数用于获取系统运行时加载、卸载类的信息。-XX:+TraceClassLoading 参数用于跟踪类加载情况，-XX:+TraceClassUnloading 用于跟踪类卸载情况。如果需要同时跟踪类的加载和卸载情况，可以同时打开这两个开关，也可以使用-verbose:class 参数。以下代码是不断加载新类、并不断卸载类的实验代码，使用参数跟踪它的类加载、卸载情况：

除了类的跟踪，JVM 还提供了-XX:+PrintClassHistogram 开关用于打印运行时实例的信息。当此开关被打开时，并且按下 ctrl+Break 按钮，就会输出系统内类的统计信息。

-XX:+DisableExplicitGC 选项用于禁止显示的 GC 操作，即禁止在程序中使用 System.gc()触发的 Full GC。通常情况下，开发人员应该对 JVM 垃圾回收机制有足够的信任，它会在恰当的时候进行垃圾回收，不需要开发者告诉它何时应该触发。如果由于开发人员的疏忽，程序中可能存在大量的 System.gc()等显示垃圾回收的代码，系统的性能就会下降。启用这个特性，可以禁用这些显示 GC，提高性能。

对应用程序来说，在绝大多数情况下，是不需要进行类的回收的。因为回收类的性价比非常低，类元数据一旦被载入，通常会伴随应用程序整个生命周期，进行类回收很可能会无功而返。当然基于 ISGI 的应用，或者使用动态字节码生成技术，大量生成动态类的应用，这些除外。

如果应用程序不需要回收类，则可以使用-Xnoclassgc 参数启动应用程序，那么在 GC 过程中，就不会发生类的回收，进而提升 GC 的性能。因此，如果读者尝试使用-XX:+TraceClassUnloading -Xnoclassgc 参数运行程序，将看不到任何输出，因为系统不会卸载任何类，所以类卸载是无法跟踪到任何信息的。

参数-Xincgc，一旦启用这个参数，系统便会进行增量式的 GC。增量式的 GC 使用特点算法让 GC 线程和应用程序线程交叉执行，从而减小应用层因 GC 而产生的停顿时间。

为确保 class 文件的正确和安全，JVM 需要通过类校验器对 class 文件进行验证。目前，JVM 中有两套校验器。JDK1.6 默认开启了新的类校验器，加速类的加载。可以使用-XX:-UseSplitVerifier 参数指定使用旧的类校验器。如果新的校验器校验失败，可以使用老的校验器再次校验。可以使用开关-XX:-FailOverToOldVerifier 关闭再次校验的功能。

在 Solaris 下，JVM 提供了几个用于线程控制的开关：

-XX:+UseBoundThreads：绑定所有用户线程到内核线程，减少线程进入饥饿状态的次数。

-XX:+UseLWPSynchronization：使用内核线程替换线程同步。

-XX:+UseVMInterruptibleIO：允许运行时中断线程。

对同样大小的内存空间，使用大页后，内存分页的表项就会减小，从而可以提升 CPU 从虚拟内存地址映射到物理内存地址的能力。在支持大页的操作系统中，使用 JVM 参数让虚拟机使用大页，从而提升系统性能。

-XX:UseLargePages：启用大页

`-XX:LargePageSizeInBytes`: 指定大页的大小。

在 64 位虚拟机上, 应用程序所占内存的大小要远远超出其 32 位版本, 约 1.5 倍左右。这是因为 64 位系统拥有更宽的寻址空间, 与 32 位系统相比, 指针对象的长度进行了翻倍。为了解决这个问题, 64 位的 JVM 虚拟机可以使用 `-XX:+UseCompressedOops` 参数打开指针压缩, 从一定程度上减少了内存的消耗。可以对以下指针进行压缩。

- Class 的属性指针 (静态成员变量)。
- 对象的属性指针。
- 普通对象数组的每个元素指针。

虽然压缩指针可以节省内存, 但是压缩和解压指针也会对 JVM 造成一定的性能损失。

7.4.16 查询 GC 命令

`-verbose`: 查询 GC 问题最常用的命令之一, 具体参数如下。

- `verbose:class`: 输出 jvm 载入类的相关信息, 当 jvm 报告说找不到类或者类冲突时可此进行诊断;
- `-verbose:gc`: 输出每次 GC 的相关情况, 后面会有更详细的介绍;
- `-verbose:jni`: 输出 native 方法调用的相关情况, 一般用于诊断 jni 调用错误信息。

`-Xprof`: 跟踪正运行的程序, 并将跟踪数据在标准输出输出; 适合于开发环境调试。

`-Xnoclassgc`: 关闭针对 class 的 gc 功能; 因为其阻止内存回收, 所以可能会导致 `OutOfMemoryError` 错误;

`-Xincgc`: 开启增量 gc (默认为关闭); 这有助于减少长时间 GC 时应用程序出现的停顿; 但由于可能和应用程序并发执行, 所以会降低 CPU 对应用的处理能力。

`-Xloggc:file`: 与 `-verbose:gc` 功能类似, 只是将每次 GC 事件的相关情况记录到一个文件中, 文件的位置最好在本地, 以避免网络的潜在问题。若与 `verbose` 命令同时出现在命令行中, 则以 `-Xloggc` 为准。

7.5 本章小结

本章首先介绍了 JVM 的内部概念, 包括内存使用、字节码组成及使用、自动内存管理等, 接下来介绍了 JVM 系统架构, 包括基本架构、初始化过程、执行引擎方式、JIT 编译器、类加载器等, 接下来重点介绍了垃圾回收相关的概念, 包括 GC、垃圾回收算法、垃圾收集器等, 最后是实践部分, 针对 JVM 的参数调整进行了着重解释和示例演示, 还对淘宝 VM 进行了一点介绍。

8 chapter

第 8 章 其他优化建议

金融衍生品 (derivatives), 是指一种金融合约, 其价值取决于一种或多种基础资产或指数, 合约的基本种类包括远期、期货、掉期 (互换) 和期权。金融衍生品还包括具有远期、期货、掉期 (互换) 和期权中一种或多种特征的混合金融工具。

程序设计的其他优化建议就好比金融衍生品, 它本身不属于 Java 基础编程技术, 有些属于延伸技术, 有些属于相关技术。不论这些优化建议属于具体哪个技术范畴, 由于它们都属于整体系统架构层面的组成部分, 所以书中会进行有针对性的讨论。

本章主要介绍和解决以下问题, 这也是本书的收官篇章:

- Java 整体发展过程及未来思路。
- 系统架构方面调优思路分享。
- Java 项目优化方式分享。
- 面向服务思维及资源隔离计数分享。
- 团队并行开发经验分享。
- 工程师性格养成。

8.1 Java 现有机制及未来发展

8.1.1 Java 体系结构变化历史

通过 20 年的发展, Java 已由一门单纯的计算机编程语言, 逐渐演变为一套强大的技术体系平台。根据不同的技术规范, Java 设计者们将 Java 划分为 3 种结构独立但却又彼此依赖的技术体系分支, 分别是 Java SE、Java EE 和 Java ME, 其中 Java EE 被广泛使用在企业级领域, 除了包括 Java API 组件外, 还扩充有 Web 组件、事务组件、分布式组件、EJB 组件、消息组件等, 整个体系框架在不断地扩大。总的来说, Java EE 也是被发展得最好的。综合 Java EE 的这些技术, 开发人员

可以构建出一个具备高性能、结构严谨的企业级应用，并且 Java EE 也是用于构建 SOA¹架构的首选平台。

对于一种有着悠久历史的编程语言来说，其历史发展过程本身既是一种财富，也是一种负担。庞大的开发者社区为这门语言的演化提供了坚实的用户基础，也为这门语言贡献了非常多的可复用的资产。从另外一方面来说，必须保持的后向兼容性也使得 Java 语言在应对变化时显得力不从心。当 Ruby 程序员可以把基本类型，如整数和浮点数，当作功能完备的对象来使用时，在 Java 世界中，基本类型和对象还是分开的。Java 程序员享受不到 Ruby 中 3.next 和 5.times 那样简洁易用的语法。虽然 Java SE5.0 引入了基本类型的自动装箱和拆箱机制，缓解了这个问题，但是依然无法实现 Ruby 那样的灵活性。当然，Java 语言在最初设计的时候，肯定权衡了各种选择的优缺点，做出了当时最合理的设计选择。通过了解一门编程语言的历史，可以深刻理解其发展过程中各种变化背后的动机。

总的来说，Java 的持续发展要感谢 Google²，正是 Google 将 Java 作为 Android³操作系统的应用层编程语言，使得 Java 可以在 PC 时代、移动互联网时代都得到迅猛发展，可以被广泛用于手持移动设备、嵌入式设备、个人电脑、高性能的集群服务器或大型机。

近几年来，Java 模块化一直是一个比较活跃的话题。所谓模块化指的是开发人员在构建大型系统时，能够将系统中的每一个功能模块进行独立的开发和物理部署，这样做的优点不仅能够有效降低各个业务模块之间的耦合，同时还能够保证当单一模块发生故障的时候不会影响系统整体的运行。当然模块化本身只是一种概念，其目的就是为了将系统中原本耦合的逻辑机型分解，以此满足各个模块之间的独立，并定义一种标准化的接口⁴契约来进行相互之间的通信。尽管 Java 目前并没有在 JDK 中内置模块化编程技术（预计 JDK9⁵会推出），但这似乎并不能阻挡开发人员选用 OSGi 技术作为首先来完成模块化编程。早在 2007 年的时候，由 Sun 公司主导并提交的 JSR-277⁶规范并没有通过 JCP⁷组织的审核，这主要是由于 JCP 专家组织通过投票将 IBM 公司提交的 JSR-291(OSGiR4.1)纳入了 Java 模块化规范标准。直到 Sun 公司在 Java7 早期时，再次提交 JSR-294（Java 模块化系统的改进支持）规范，可惜最终还是未能如愿。不得已 Sun 公司只能避开 JCP 组织，在 OpenJDK 中创建了一个叫作 Jigsaw 的子项目来实现 Java 模块化编程技术，但该项目却因为一些原因被迫延期到 Java9 中进行发布。不管技术如何发展，我们可以断定 OSGi 技术会使 Java 模块化规范标准。

纵观 Java 语言的发展历史。

¹ 面向服务的体系结构是一个组件模型，它将应用程序的不同功能单元（称为服务）通过这些服务之间定义良好的接口和契约联系起来。

² Google（中文名：谷歌），是一家美国的跨国科技企业，致力于互联网搜索、云计算、广告技术等领域，开发并提供大量基于互联网的产品与服务，其主要利润来自于 AdWords 等广告服务。

³ Android 是一种基于 Linux 的自由及开放源代码的操作系统，主要使用于移动设备，如智能手机和平板电脑，由 Google 公司和开放手机联盟领导及开发。

⁴ 通过接口可以让两个不相关的软件服务互相访问对方。

⁵ OpenJDK9 12 月 10 日完成属性开发，Oracle JDK 预计 2016 年 3 月对外发布。

⁶ 即 Java 模块化系统需求。

⁷ JCP（Java Community Process）是一个开放的国际组织，主要由 Java 开发者以及被授权者组成，职能是发展和更新。

1991 年, James Gosling⁸、Mike Sheridan 和 Patrick Naughton⁹在 Sun 公司开始了一种新的编程语言的设计和开发工作。这种新的语言换过好几个名字, 从最初的 Oak 到后来的 Green, 直到最后被定名为 Java。在最初的时候, 这门语言是以与计算机相连接的智能消费电子产品作为目标平台而设计的。Java 语言最早运行在数字有线电视的控制器上, 不过, 这项技术对当时的产业来说过于超前了。正当发展遇到瓶颈的时候, 幸运的是, Java 赶上了从 1995 年开始的互联网发展的上升期。1995 年底, Sun 公司正式发布 Java 语言。在随后不久的 1996 年初, JDK1.0 发布。当时 Java 语言的杀手级应用是 Java Applet。当时主流的浏览器都支持 Applet。在当时, HTML 语言所能提供的交互性能力还比较弱, 而用户对于互联网应用的热情却非常高涨, 这就在 Web 应用的交互能力和用户的需求之间产生了一个缺口。Java Applet 的出现正好填补了这个空白。Applet 提供了丰富的用户界面组件, 以及 2D 图形绘制能力, 其所提供的交互性体验远优于当时的 HTML, 因此 Java Applet 得到了广泛流行。Java Applet 的代码是从远程服务器上下载到用户的本地浏览器中来运行的。这种需求催生了 Java 语言的一个重要特性, 就是类加载。动态类加载的概念被认为是 Java 语言唯一的重要创新。

在 JDK1.0 之后, JDK1.1 于 1997 年发布。在这个版本中, 一些新特性被加入进来, 包括内部类、JavaBeans、JDBC、RMI 和反射等。JDK1.1 发布三个星期之后, 就达到了 22 万的下载量。

从 JDK1.1 开始, Java 为不同的目标平台提供了不同的配置, 这就是标准版、企业版和移动版。Sun 公司也启用了 Java2 这样一个新名称。这些不同的配置也被对应地称为 Java2 标准版 (J2SE)、Java2 企业版 (J2EE)、Java2 移动版 (J2ME)。J2SE1.2 于 1998 年发布, 其中最重要的更新是添加了 Swing 用户界面库和集合类框架。Swing 用户界面的引入, 使 Java 在桌面应用开发中占据了一席之地。而 Java 的集合类框架则一直被认为是架构和 API 设计的良好典范, 其主要设计和开发者是《Effective Java》的作者 Joshua Bloch¹⁰。

Java2 的版本更新以较快的速度进行。2000 年发布的 J2SE1.3 中最重要的改进是使用 HotSpot 作为默认的 Java 虚拟机, 极大地提升了 Java 程序的运行性能, 在一定程度上缓解了一直以来为开发者所诟病的性能问题。动态代理机制也在这个版本中被印日, 允许以简单的方式来实现面向方面编程 (AOP)。

随着 Java 语言的不断发展和流行, 对 Java 平台的需求也日益增多。如何让 Java 语言健康有序地发展, 成为 Sun 公司需要解决的一个问题。Sun 公司尝试向国际标准化组织靠拢, 以寻求 Java 语言的标准化。后来 Sun 公司放弃了这种努力, 转而拥抱广大的开发者社区, 采用社区驱动的方式来促进 Java 语言的发展。1998 年, Java 程序社区 (Java Community Process, JCP) 成立, 其中包含了对 Java 感兴趣的公司、研究机构和个人。对 Java 平台的改进都通过 JCP 来运作。对 Java 平台所做的每一个修改都以 Java 规范请求 (Java Specification Request, JSR) 的形式来提交给 JCP 进行审阅和批准, 最终成为 Java 平台的一部分。

2002 年, 首个采用 JCP 方式开发的 Java 平台 J2SE1.4 发布。在这个版本中, 更多的类库被加

⁸ 詹姆斯·高斯林 (James Gosling, 1955 年 5 月 19 日—, 出生于加拿大), 软件专家, Java 编程语言的共同创始人之一, 一般公认他为“Java 之父”。

⁹ 帕特里克·诺顿(Patrick Naughton)诺顿是 Sun Java 项目的中坚人物, 并曾在迪士尼担任互联网业务高管。

¹⁰ 卡内基梅隆大学计算机专业博士, Google 的 Java 首席架构师。

入进来,包括正则表达式、非阻塞 I/O (NIO)、日志记录 API、XML 和 XSLT 支持以及安全和加密功能等。

2004 年发布的 J2SE5.0 是 Java 语言发展历史上的一个重要的版本。在这个版本中,Java 增加了很多语法上的新特性,包括泛型支持、注解、基本类型的自动装箱和拆箱、枚举类型、参数长度可变的方法、增强的 for 循环和静态引入等。而在类库方面,并发实用类库 `java.util.concurrent` 的引入,极大地降低了并发应用开发的复杂度。在 Java5 版本的时候,Java 设计者们通过 JSR-166 的规范制定,在 `java.util.concurrent` 包下为开发人员提供了基于粗粒度的多核并行计算框架,只不过这种基于粗粒度的并行计算模式,并不能在处理效率上达到令人满意的程度。因为这种基于粗粒度的并行计算,根本无法高效组合所有可用物理核心一起进行并行任务处理,甚至在某些情况下,还有可能导致部分物理核心处于空闲的状态。所以 Java7 在 `java.util.concurrent.forkjoin` 包下新增了基于细粒度的多核并行计算 Fork/Join 框架。该框架的设计初衷是将一个任务量化到最小,并提供高计算密度的并行处理性能。简单来说,我们可以将一个任务拆分成若干个子任务,直到这个任务足够小,然后每一个子任务被独立并行计算,直到任务执行完成,再将其逐个合并为一个完整的任务,这就是 Fork/Join 框架提供的细粒度并行计算模式,我们在第 5 章已经详细介绍了。

Java SE6 于 2006 年发布,这个版本的新特性主要体现在对脚本语言的支持、Java 编译器 API 和可插拔式注解等方面。Java SE6 另一个重要提升是在性能方面,包括核心平台和 Swing 用户界面库的性能都有了很大的提升。在正式版本发布之后,每隔一段时间就会有很小的更新修订版本发布。

除了 Java7 能够高效地利用 Fork/Join 框架实现多核并行计算外,开源基金会 Apache 提供的 Hadoop MapReduce 框架也是一个高效的海量数据计算框架,它允许开发人员将其部署在廉价的集群服务器上处理 TB 级别的数据。当然还有一些编程语言自诞生那天起,就是为了解决并行计算而来,比如 Scala、Clojure 和 Erlang 等。这类编程语言,不仅继承了面向对象的特性,同时还结合了函数式编程¹¹等特性。虽然目前 Java 同样也能够使用函数式编程,但代码将会显得非常冗余,不过 Java 设计者们在 Java8 中提供对 Lambda 的支持,这极大地改善了 Java 对于函数式编程的不足。

Java7 提出了全新的文件系统 NIO2.0,利用 NIO2.0,开发人员无须关注 I/O 细节,因为新文件系统中封装有大量的通用操作,便于开发人员更好地关注自身业务。并且在 I/O 模型方面,将支持调用操作系统的 IOCP (Input/Output Completion Port, 输入/输出完成端口) 接口实现真正的异步 I/O,尽可能避免因 I/O 问题导致的系统瓶颈出现。

Java 不断发展的过程中,操作系统也在不断地发展。相对于传统的 32 位虚拟机,64 位虚拟机所具备的最大优势就是可以访问大内存,32 位虚拟机做的最大可用内存空间被限定在了 4GB,并且 Java 堆区的大小如果是在 Windows 平台下最大只能设置到 1.5GB,而在 Linux 平台下最大也只能设置到 2GB-3GB 的上限,也就是说,Java 堆区的内存大小设置还需要依赖于具体的操作平台。既然 32 位虚拟机无法满足大内存消耗的应用场景,那么 64 位虚拟机的出现则是顺理成章的,64

¹¹ 函数式编程是种编程典范,它将电脑运算视为函数的计算。函数编程语言最重要的基础是 λ 演算 (lambda calculus)。而且 λ 演算的函数可以接受函数当作输入 (参数) 和输出 (返回值)。和指令式编程相比,函数式编程强调函数的计算比指令的执行重要。和过程化编程相比,函数式编程里,函数的计算可随时调用。

位虚拟机之所以能够访问大内存，是因为其采用了 64 位的指针架构，这也是寻址访问大内存的关键要素。

在 JDK1.6 Update14 版本之前，64 位虚拟机的综合性能表现实际上是不如 32 位虚拟机的，这主要是因为 OOPS (Ordinary Object Pointers, 普通对象指针) 从 32 位膨胀到 64 位后，CPU Cache Line 中的可用 OOPS 变少，这样一来就会直接影响并降低 CPU 的缓存使用率，这就是 64 位虚拟机在性能上之所以落后于 32 位虚拟机的主要原因。其次由于部署在 64 位虚拟机上的性能都需要用到大内存，尤其是互联网项目，经常需要使用多达几十乃至几百 GB 的内存，这对于传统的 32 位虚拟机将无法承载，只能依靠 64 位虚拟机去支撑。但是管理这么大的内存开销对于 GC 来说将会是一场非常严峻的考验，甚至很有可能去导致 GC 在执行内存回收期间消耗更长的时间，同时也意味着工作线程的等待时间将会延长。随着如今 64 位虚拟机的逐渐成熟，指针压缩将会通过对齐补白等操作将 64 位指针压缩为 32 位，以此改善 CPU 缓存使用率达到提升 64 位虚拟机运行性能的目的。

淘宝的技术团队对 Java 虚拟机的优化工作其实早已不是停留在简单的参数调整上面，而是充分结合了企业自身的业务特点以及实际的应用场景，在 OpenJDK 的基础之上通过修改大量的 HotSpot 源代码，深度定制了淘宝专属的高性能 Linux 虚拟机 TAOBAOVM。从严格意义上来说，在提升 Java 虚拟机性能的同时，严重依赖于物理 CPU 类型。也就是说，部署有 TAOBAOVM 的服务器中，CPU 全都是清一色的 Intel CPU，且编译手段采用的是 Intel C/CPP Compiler 进行编译，以此对 GC 性能进行提升。除了优化编译效果外，TAOBAOVM 还使用 crc32 指令实现 JVM intrinsic 降低 JNI 的调用开销。

除了在性能优化方面下足了功夫，TAOBAOVM 还在 HotSpot 的基础之上大幅度扩充了一些特定的增强实现，比如创新的 GCIH (GC invisible heap) 技术实现 off-heap，这样一来就可以将生命周期较长的 Java 对象从 heap 中移至 heap 之外，并且 GC 不能管理 GCIH 内部的 Java 对象，这样做最大的好处就是降低了 GC 的回收频率以及提升了 GC 的回收效率，并且 GCIH 中的对象还能够多个 Java 虚拟机进程中实现共享。其他补充技术还有利用 PMU hardware 的 Java profiling tool 和诊断协助功能等。

8.1.2 Java 语言面临的挑战

随着各种新的需求和应用场景的不断出现，新的软件开发思想和程序设计语言也层出不穷。虽然 Java 语言多年来一直稳坐编程语言排行榜的头把交椅，但是它近年来也面临着其他语言的冲击。各种唱衰 Java 的悲观论调在社区里此起彼伏。Java 是静态强类型语言。这种特性使 Java 编译器在编译时可以发现非常多的类型错误，而不会让这些错误在运行时才暴露出来。对于构建一个稳定而安全的应用来说，这是一个很大的优势，但是这种静态的类型检查也限制了开发人员编写代码时的创造性和灵活性。

首先是因为互联网的发展带来的各种动态语言，动态语言的灵活性带来了开发效率的极大提升，Java 语言的静态类型特点限制了开发人员编码时的创造性与灵活性；其次是 Groovy¹²、

¹² 一种基于 JVM (Java 虚拟机) 的敏捷开发语言，它结合了 Python、Ruby 和 Smalltalk 的许多强大的特性，Groovy 代码能够与 Java 代码很好地结合，也能用于扩展现有代码。由于其运行在 JVM 上的特性，Groovy 可以使用其他 Java 语言编写的库。

Scala¹³、JRuby¹⁴等运行在 Java 虚拟机上的语言，它们既有简洁优雅的语法，又能充分利用 Java 虚拟机的资源，极具竞争力；再者，Java 在云计算时代也面临以 Go 语言¹⁵为主的容器（Docker¹⁶等技术）生态圈的挑战。

Web 2.0 概念的出现和互联网应用的火热，为新语言的流行创造了契机。Ruby¹⁷语言凭借 Ruby on Rails 一举走红，Google 的 Web 应用开发平台 Google App Engine 最初也只支持 Python¹⁸一种语言，甚至流行的 JavaScript 语言也借助于 Node.js¹⁹和 Jaxer²⁰等平台在服务器端开发中占据了一席之地。这些语言的共同特征是代码动态型与拥有灵活自由的语法。开发人员一旦掌握了这些语言，开发效率会非常的高。在这一点上，Java 语言纷繁的语法就显得缺乏吸引力。Java 语言也受到同样运行在 Java 虚拟机上的其他语言的挑战。这些语言包括 Groovy、Scala、JRuby 和 Jython 等。任何语言，只要它生成的字节代码符合 Java 字节代码规范，就可以在 Java 虚拟机上运行。前面提到的这些 Java 虚拟机上的语言既具有简洁优雅的语法，又能充分利用已有的 Java 虚拟机资源，相对于 Java 语言本身而言，非常具有竞争力。

Java 平台的不足之处是整个 Java 平台的复杂性。最早在 JDK1.0 发布的时候只有几百个 Java 类，而到 Java6 时，已经包括 Java SE、Java EE 和 Java ME 等多个版本，所包含的 Java 类多达数千个。对于普通开发者来说，完全理解和熟悉如此庞大的难度非常大。在日常的开发过程中，经常可以看到开发者在重复实现某些功能，而这些功能在 Java 类库中已经存在，只是不被人知道而已。除了庞大的类库之外，Java 语言的语法本身也缺乏足够的灵活性，实现某些功能所需的代码量可能是其他语言的几倍。另外一个复杂性体现在 Web 应用开发方面。一个完整的 Java EE 应用程序要求程序员掌握和理解的概念太多，要使用的库也非常多，过去管理整个 Java EE 应用程序相关库文件的方式非常复杂，需要自己建立文件夹专门用来存储类库文件，后来出现了 ANT²¹和 Maven²²，开始慢慢让管理变得容易起来了。这一点我们从市面上到处可见的以 Java Web 应用开发和 Struts、Spring、Hibernate 等框架为内容的图书上就可以看出来。虽然新出现的 Grails 和 Play 框架等都试图降低这个复杂度，但是这些新的框架的流行仍然需要足够长的时间。

其实 JVM 也是一种容器，但是这种容器特性正在被 Linux 学习与赶超，那么，JVM 的定位就可能比较尴尬。Docker 之类容器可以在本地笔记本或电脑上运行，然后同样可以部署到云上运行。

¹³ 一门多范式的编程语言，一种类似 Java 的编程语言，设计初衷是实现可伸缩的语言、并集成面向对象编程和函数式编程的各种特性。

¹⁴ 一个采用纯 Java 实现的 Ruby 解释器，由 JRuby 团队开发。它是一个自由软件，在 CPL/GPL/LGPL 三种许可协议下发布。

¹⁵ 谷歌 2009 发布的第二款开源编程语言。Go 语言专门针对多处理器系统应用程序的编程进行了优化，使用 Go 编译的程序可以媲美 C 或 C++ 代码的速度，而且更加安全、支持并行进程。

¹⁶ 一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。

¹⁷ 一种为简单快捷的面向对象编程（面向对象程序设计）而创的脚本语言。

¹⁸ 一种面向对象、解释型计算机程序设计语言。

¹⁹ Node.js 是一个基于 Chrome JavaScript 运行时建立的平台，用于方便地搭建响应速度快、易于扩展的网络应用。Node.js 使用事件驱动，非阻塞 I/O 模型而得以轻量 and 高效，非常适合在分布式设备上运行的数据密集型的实时应用。

²⁰ 世界上第一个真正的 Ajax 服务器，服务器端和客户端都是使用 JavaScript，而且可以相互调用。

²¹ 一种基于 Java 的 build 工具。理论上来说，它有些类似于（UNIX）C 中的 make，但没有 make 的缺陷。

²² 基于项目对象模型(POM)，可以通过一小段描述信息来管理项目的构建，报告和文档的软件项目管理工具。

当在云上运行时，Kubernetes²³能够以一种可控的方式升级容器从而实现运行管理一批容器，如同一个大型船队或舰队一样，你可以控制它们的流量访问量，可以指定多少个容器来扩展支撑一个服务的运行，随着访问量提升，你通过增加容器数量能够整个系统的负载能力。

当然，Java 的大型分布式系统越来越多，Java 在云计算与分布式系统中还是扮演主要角色，形成一个大型的生态圈。当我们站在泰山之上，一览众山小，当你在全球拥有多个数据中心时，语言已经变得不那么重要了，关键是架构设计。

Go 语言相对 Java 来说，主要优点是其并发组件模型，Java 的并发比较低级，无非是多线程与锁，想搞清楚 Java 中各种锁的用途，包括数据集合 Collection 的线程安全性与性能差异对比，需要花费大量时间与精力，包括具体的使用经验。而 Go 语言使用了 Channel/CEP 这样的组件简单封装了多线程与锁，将以前 JMS 的 Queue 队列模型架构引入到了语言之中，两个对象之间交互只要通过 Channel 通道就可以。这种模型保证了并发性，又简化了编程模型，无疑受到很多人的欢迎。

Go 语言当前也受到更加强劲的 Rust²⁴语言挑战，如果说，Go 语言的 Channel 是一种有形的设计，那么，Rust 语言的并发模型达到无形的设计，只要你编写好函数方法，安全性与并发性就无形中得到了解决，不用专门去思考并发，有意识地去使用并发组件模型编程。现在的开发语言如雨后春笋，主要原因是 CPU 进入多核并发时代，以及大型架构进入分布式系统，如何使用一种语言从微观的 CPU 多核之间并发到数万台服务器之间的分布式计算处理，这种大一统的愿景促使人们在不断探索。在 Java 中，我们可以通过框架来实现这点，以 Jdon 框架为例，虽然能够勉强实现并发与分布式，但是这种实现需要很强的知识背景，不利于初学者上手。而要达到普及这个目标，必须从语言入手，让语言初学者在学习掌握语言以后，无形中就会实现了并发与分布式。

Go 语言在这方面比较突出，其并发模型以读写操作为基础。请注意，Java 等语言并发模型没有这么高，它们是以线程为基础，再应用到读写场景中，而我们现实中必须以读写为基础，再应用到具体业务场景中，这里面高低层次：线程 -> 读写操作 -> 业务应用，无疑越靠近业务应用的语言越能简化我们的开发，而分布式系统也是基于读写操作，著名的 CAP 定理也隐含了以读写操作为基础的语境。

8.1.3 Java8 的新特性

除了 Lambda 表达式和模块化支持之外，Java SE8 种的其他更新内容包括：

- 把 JRockit 虚拟机中的部分特性整合到 HotSpot 虚拟机中，提供一个统一的虚拟机实现；
- 集成 JavaFX3.0。在 Java SE8 中会直接集成 JavaFX3.0；
- 在虚拟机上可以直接使用新的 JavaScript 引擎，以及更好的 JavaScript 和 Java 代码之间的互操作性。新的 JavaScript 引擎被称为 Nashorn，是一个基于 JSR292 的实现；
- 在移动设备商，增加对多点触控、摄像头、地理位置信息、罗盘和重力加速器的支持；

²³ Google 开源的容器集群管理系统。它构建 Ddocker 技术之上，为容器化的应用提供资源调度、部署运行、服务发现、扩容缩容等整套功能，本质上可看作是基于容器技术的 mini-PaaS 平台。

²⁴ Mozilla 开发的注重安全、性能和并发性的编程语言。

- 对 Java 安全、网络、国际化和可访问性 API 的更新;
- 允许 Java 中的注解出现在类型的任意使用方式上,而不仅限于类型、方法、域和变量的声明中。注解可以使用的位置包括方法调用的接收者、泛型类型参数、数组、强制类型转换、instanceof 操作符、对象创建、泛型中类型参数的上界和下界、类继承关系和 throws 子句。具体的细节由 JSR308 (Annotations on Java Types) 规范来描述;
- 新的日期和时间 API,用来解决当前使用 java.util.Date 类和 java.util.Calendar 类处理日期和时间中产生的问题。具体的细节由 JSR310 (Date and Time API) 规范来描述。

8.1.4 Java 语言前景

Java 平台正沿着三个重要方向发展:

第一,提高开发人员的生产效率,尽可能在保证健壮性的同时降低语法的复杂度。由于 Java 语言的静态强类型特性,使用 Java 语言编写的程序代码一半比较烦琐,包含了过多不必要的语法元素,这在一定程度上降低了开发人员的生产效率。大量的时间被浪费在语言本身上,而不是真正需要的业务逻辑上。从另外一个角度来说,Java 语言的这种严谨性,对于复杂应用的团队开发是大有好处的。有利于构建健壮的应用。Java 语言需要在这两者之间达到一个平衡。Java 语言的一个发展趋势是在可能的范围内降低语言本身的语法复杂度。从 J2SE 5.0 种增强的 for 循环,到 Java7 的 try-with-resources 语句和 <> 操作符,再到 Java8 引入的 Lambda 表达式,Java 正在不断地简化自身的语法。

第二,提高平台的性能,新的 Java 技术要能让 Java 应用充分利用硬件升级(多核 CPU 和多 CPU 架构)所带来的性能提升。Java 平台的性能一直为开发人员所诟病,这主要是因为 Java 虚拟机这个中间层次的存在。随着硬件技术的发展,越来越多的硬件平台采用了多核 CPU 和多 CPU 的架构。应用程序应该充分利用这些资源来提高程序的运行性能。Java 平台需要帮助开发人员更好地实现这个目标。Java7 中的 fork/join 框架是一个高效的执行框架。Java8 对集合类框架和相关 API 做了增强,以支持对批量数据进行自动的并行处理。

第三,模块化,通过把 Java 平台提供的类库划分为相互依赖的不同模块,不仅可以提升程序员的开发效率,而且还能提升应用的运行速度。一直以来,Java 平台所包含的各种功能不同的类库是一个统一的整体。在一个程序的运行过程中,很多类库其实是不需要的。比如对于一个服务器端运行的程序来说,Swing 用户界面组件库通常是不需要的。模块化的含义是把 Java 平台提供的类库划分成不同的相互依赖的模块,程序可以根据需要选择运行时所依赖的模块,只有被选择的模块才会在运行时被加载。模块化的实现不仅可以应用到 Java 平台本身,也可以应用到 Java 应用程序的开发中,OpenJDK 中的 Jigsaw 项目提供了这种模块化的支持,未来在 Java9 中 Osgi 会负责提供这样的模块化支持。

对于 Java 语言的未来,有理由相信 Java 平台会一直发展下去。其中很重要的依据是 Java 平台的开放性。依托 JCP 和 OpenJDK 项目,Java 平台不仅在语言规范这个层次上有健康的开放管理流程,也有与之对应的参考实现。Java 语言有着人数众多的开发者社区,每年有非常多新的开发者学习和使用 Java。大量的开发者使用 Java 语言开发各种不同类型的应用。在社区中可以看到很多提供不同功能的类库和框架。Java 虚拟机已经被安装到数以十亿计的不同类型的设备商,包括服务器、个人计算机、移动设备和智能卡等。依托庞大的社区和数量众多的运行平台,Java 语言

的发展前景是非常乐观的。Java7、Java8，以及即将在 2016 年发布的 Java9，有力地回应了这样的论调，让开发者取舍看到了 Oracle 执掌 Java 后 Java 的发展前景，给了开发者信心。

8.1.5 物联网：Java 和你是一对

从 1969 年至今的这段漫长时光当中，网络设备已经完成了完整的爆发式增长。从当初通过 ARPANET 实现对接的四台高校计算机开始，如今世界上已经有二十亿人频繁访问互联网。在不久的将来，联网设备数字还将迅速翻番甚至再次翻番，即由目前的数十亿台增长至嵌入式处理时代的成百上千亿台。我们生活中的方方面面都将与联网设备相结合：家庭环境、办公环境、车载环境、设备、工具以及玩具等。可以预见，作为一款专门针对嵌入式计算与实时化流程场景所构建的编程语言，Java 将成为物联网时代下的最佳选择。

十九年前，David L. Ripps 曾为 JavaWorld 编写了一份概述性资料，介绍了 Java 在嵌入式系统中的作用。Ripps 的文章从今天的角度来看同样极具可读性，特别是对于那些希望了解嵌入式系统编程接口如何与联网移动设备及物联网机制协作的朋友而言。

尽管物联网浪潮的席卷之势中确实存在一部分炒作成分，但其背后的现实情况在于，互联网增长将使上一代计算机变得相对比较琐碎。物联网不仅客观存在，而且还将给一切带来颠覆性的改变。参考以下时间进程，我们首先对过往互联网技术在不同阶段中的发展轨迹作出一番回顾：

- 1982 年到 1989 年：TCP/IP 网络诞生。
- 1985 年到 1989 年：互联网技术的商业化趋势开始出现。
- 1990 年到 1991 年：万维网正式建立。
- 1990 年到 1998 年：传统台式计算机被重新设计为实质层面上的联网设备。
- 1996 年至今虽然进展缓慢但却可以肯定的是，我们正逐步进入到移动联网设备（即物联网）主导一切的新时代当中。

目前作为物联网前提性条件各类补充性技术正在陆续上线。HTTP/2 是一套关键性网络协议，它的出现在一定程度上实现了机器到机器之间的通信需求。Thingsee 则是开发者工具领域的典型代表，也标志着物联网发展所需要的硬件基础正逐渐成形。

硅谷智囊 Tim O'Reilly 已经作出强调，表示物联网的成果将不仅仅是将咖啡机或者电冰箱等无关紧要的设备接入网络那么单纯。在理想的传感器与自动化机制支撑之下，物联网将真正将人类文明提升到新高度。而 Java 将在将在这场颠覆性变革中扮演主力角色。

2014 年 9 月，Andrew C. Oliver 撰写了一篇关于物联网实现水平与团队协作间关系的文章。在这种情况下，团队协作体系将由人与计算机共同构成。

由于设备的通信对象不再局限于人、同时需要面对其他设备，因此将从根本层面带来一系列新功能。具体而言，我们的电冰箱不仅能够感知到用户的西红柿储量即将告罄，同时也能根据个人饮食习惯发出食品订单。普适计算的成功也恰恰体现在这里，其融入背景当中，并与其他设备共同完成任务、事件以及对接。只有执行级别的结果才会被交付至使用者面前。物联网的崛起将带来大量我们前所未见、甚至难以想象的创新型成果，并以无缝化方式将其奉至我们手中。

Java 在诞生早期正是针对嵌入式计算而打造。其早期版本专门针对各类家用电器，如电视机

顶盒接口。James Goslin 在打造 Java 初始版本时正是将设备间通信作为其关注重点，而他当时就想到其作用不仅要实现设备与人之间的通信、更要承担起设备与设备间的通信任务。二十年之后，这些初始设计优势终于迎来了自己的黄金时代，物联网的光辉岁月即将拉开序幕。

出色的普及水平也使其适合物联网时代的实际需求。全球范围内投入到 Java 领域的海量资源使这款编程语言成为新生代程序员们的最爱，同时也确保了其能够在全部以其为基础的生产系统中得到良好的维护与支持。数十万款成功的应用程序及系统方案已经充分证明了 Java 的强大实力。

对于希望在嵌入式编程领域一展身手的开发人员而言，最重要的是对 Java 平台中的不同组成部分加以区分。在嵌入式编程工作中，我们无须对自己的编码或者阅读方式做出任何变更：出色的 Java 程序员能够像查看典型桌面企业应用程序那样轻松阅读嵌入式源代码内容。不过库，特别是在开发（及测试）环境中，将专门面向嵌入式 Java 编程。请确保大家拥有适用于目标嵌入式环境的正确工作链。

甲骨文公司已经针对实时系统对 Java SE 进行了改进，并表示 Java SE 较过去已经能够带来更理想软实时要求支持效果。此外，Java 对于嵌入式编程作出了多项承诺，而且其在满足即将全面爆发之物联网的需求及可能性方面还有很长的发展道路要走。数百亿由 Java 驱动的设备将在未来几年内成为物联网网络体系中的组成部分。

8.1.6 Java 模块化发展

近几年来，Java 模块化一直是一个比较活跃的话题。所谓模块化指的就是开发人员在构建大型系统时，能够将系统中的每一个功能模块进行独立的开发和物理部署，这样做的优点不仅能够有效降低各个业务模块之间的耦合，同时还能够保证当单一模块发生故障时不会影响系统整体的运行。当然模块化本身只是一种概念，其目的就是为了将系统中原本耦合的逻辑机型分解，以此满足各个模块之间的独立，并定义一种标准化的接口契约来进行相互之间的通信。尽管 Java 目前并没有在 JDK 中内置模块化编程技术（预计 JDK9 会推出），但这似乎并不能阻挡开发人员选用 OSGi 技术作为模块化编程的首选。早在 2007 年的时候，由 Sun 公司主导并提交的 JSR-277（Java 模块化系统）规范并没有通过 JCP 组织的审核，这主要是由于 JCP 专家组织通过投票将 IBM 公司提交的 JSR-291（OSGiR4.1）纳入了 Java 模块化规范标准。直到 Sun 公司在 Java7 早期时，再次提交 JSR-294（Java 模块化系统的改进支持）规范，可惜最终还是未能如愿。不得已 Sun 公司只能避开 JCP 组织，在 OpenJDK 中创建了一个叫作 Jigsaw 的子项目来实现 Java 模块化编程技术，但该项目却被迫延期到 Java9 中进行发布。可以断定 OSGi 技术会使 Java 模块化规范标准。

模块化是个一般概念，这一概念也适用于软件开发，可以让软件按模块单独开发，各模块通常都用一个标准化的接口来进行通信。实际上，除了规模大小有区别外，面向对象语言中对象之间的关注点分离与模块化的概念基本一致。通常，把系统划分为多个模块有助于将耦合减至最低，让代码维护更加简单。

类库毫无疑问也是模块。对于类库来讲，可能没有一个单一接口与之通信，但往往却有‘public’ API（可能被用到）和‘private’ package（文档中说明了其用途）。此外，它们也有自己依赖的类库（比如 JMX 或 JMS）。这将引起自动依赖管理器引入许多并非必需的类库：以 Log4J-1.2.15 为例，引入了超过 10 个依赖类库（包括 javax.mail 和 javax.jms），尽管这些类库中有不少对于使用 Log4J 的程序来说根本不需要。

某些情况下，一个模块的依赖可以是可选的；换句话说，该模块可能有一个功能子集缺少依赖。在上面的例子中，如果 JMS 没有出现在运行时 classpath 中，那么通过 JMS 记录日志的功能将不可用，但是其他功能还是可以使用的。Java 通过使用延迟链接——deferred linking 来达到这一目的：直到要访问一个类时才需要其出现，缺少的依赖可以通过 `ClassNotFoundException` 来处理。其他一些平台的弱链接，例如 weak linking 概念，也是做类似的运行时检查。

通常，模块都附带一个版本号。许多开源项目生成的发行版都是以类似 `log4j-1.2.15.jar` 的方式命名的。这样开发者就可以在运行时通过手动方式来检测特定开源类库的版本。可是，程序编译的时候可能使用了另一个不同版本的类库：假定编译时用 `log4j-1.2.3.jar` 而运行时用 `log4j-1.2.15.jar`，程序在行为上依然能够保持兼容。即使升级到下一个小版本，仍然是兼容的（这就是为什么 `log4j 1.3` 的问题会导致一个新分支 `2.0` 产生，以表示兼容性被打破）。所有这些都是在基于惯例而非运行时的已知约束。

无论在编译时还是运行时，Java 的 classpath 都是扁平的。换句话说，应用程序可以看到 classpath 上的所有类，而不管其顺序如何（如果没有重复，是这样；否则，总是找最前面的）。这就使 Java 动态链接成为可能：一个处于 classpath 前面的已装载类，不需要解析其所引用的可能处于 classpath 后面的那些类，直到确实需要他们为止。

如果所使用的接口实现到运行时才能清楚，通常使用这种方法。例如，一个 SQL 工具可以依赖普通 JDBC 包来编译，而运行时（可以有附加配置信息）可以实例化适当的 JDBC²⁵ 驱动。这通常是在运行时将类名（实现了预定义的工厂接口或抽象类）提供给 `Class.forName` 查找来实现。如果指定的类不存在（或者由于其他原因不能加载），则会产生一个错误。

因此，模块的编译时 classpath 可能会与运行时 classpath 有些微妙的差别。此外，每个模块通常都是独立编译的（模块 A 可能是用模块 C 1.1 来编译的，而模块 B 则可能是用模块 C 1.2 来编译的），而另一方面，在运行时则是使用单一的路径（在本例中，即可能是模块 C 的 1.1 版本，也可能是 1.2 版本）。这就会导致依赖地狱（Dependency Hell），特别当它是这些依赖传递的末尾时更是这样。不过，像 Maven 和 Ivy 这样的构建系统可以让模块化特性对开发者是可见的，甚至对最终用户也是可见的。

目前，Java 领域存在许多模块化系统和 plugin 体系。IDE 是名气最大的，IntelliJ、NetBeans 和 Eclipse 都提供了其自己的 plugin 系统作为其定制途径。而且，构建系统（Ant、Maven）甚至终端用户应用（Lotus Notes、Mac AppleScript 应用）都有能够扩展应用或系统核心功能的概念。

OSGi 是 Java 领域里无可辩驳的最成熟的模块系统，它与 Java 几乎是如影相随，最早出现于 JSR 8，但是最新规范是 JSR 291。OSGi 在 JAR 的 MANIFEST.MF 文件中定义了额外的元数据，用来指明每个包所要求的依赖。这就让模块能够（在运行时）检查其依赖是否满足要求，另外，可以让每个模块有自己的私有 classpath（因为每个模块都有一个 `ClassLoader`）。这可以让 dependency hell 尽早被发现，但是不能完全避免。和 JDBC 一样，OSGi 也是规范（目前是 4.2 版），有多个开源（及商业）实现。因为模块不需要依赖任何 OSGi 的特定代码，许多开源类库现在都将其元信息嵌入到 manifest 中，以便 OSGi 运行时使用。有些程序包没有这么做，也可以用 `bundle`

²⁵ JDBC（Java Data Base Connectivity，Java 数据库连接）是一种用于执行 SQL 语句的 Java API，可以为多种关系数据库提供统一访问，它由一组用 Java 语言编写的类和接口组成。

这样的工具,它可以处理一个已有的 JAR 文件并为其产生合适的默认元信息。自 2004 年 Eclipse 3.0 从专有 plugin 系统切换到 OSGi 之后,许多其他专有内核系统(JBoss、WebSphere、Weblogic)也都随之将其运行时转向基于 OSGi 内核。

OpenJDK 中的 Jigsaw 项目的目标在于为 Java 平台增加模块化的支持,这也是 Java SE8 的重要组成部分。

Jigsaw 项目为 Java 语言增加了模块的概念。模块式 Java 类型的集合。每个模块由自己的名称、一个可选的版本号,以及当前模块和其他模块之间关系的描述。模块之间最重要的关系是依赖关系。一个模块可以依赖其他的模块,并利用所依赖的模块提供出的 Java 类。对于一个模块来说,模块本身的信息由名为 module-info.java 的 Java 文件来提供。下面给出了 module-info.java 文件的源代码。模块“com.my.base”的版本号是 1.0,依赖于版本号为 2.0 的“com.example”模块。通过“exports”声明了模块“com.my.base”中的“com.my.base.utils”包对依赖它的模块是可见的。通过“class”声明了模块“com.my.base”的主 Java 类是“com.my.base.MainApp”。如代码清单 8-1 所示。

代码清单 8-1 Map cacheMap 缓存池

```
module com.my.base @ 1.0{
    requires com.example @ 2.0;
    exports com.my.base.utils;
    class com.my.base.MainApp;
}
module com.my.ui @ 1.0{
    requires com.my.base;
}
```

在模块中,除了 module-info.java 文件之外,其他的内容与一般的 Java 程序中可以包含的内容相同,可以包括 Java 源代码、资源文件、配置文件和本地代码等。在编译过程中,模块中包含的源代码会被变异,然后进行打包和发布。打包好的模块可以安装到某个模块仓库中。这个仓库中包含了所有可用的模块。在运行时,虚拟机会负责从模块仓库中加载模块,并与它所依赖的其他模块进行链接,之后就可以运行该模块了。

8.1.7 OpenJDK 的发展

在 Java 语言产生之后的很长一段时间内,Java 语言的规范制定以及类库和运行环境的开发工作,都是由 Sun 公司独立完成的,都是 Sun 公司的私有实现。在 Java 发展的初期,这种方式避免了烦琐的流程,降低了对外的沟通成本,是有利于 Java 语言的快速发展的。但是随着 Java 语言的日益流行,这种 Sun 公司的私有实现模式已经不能适应发展的要求了。首先是广大 Java 开发者对 Java 的需求越来越多,单凭 Sun 公司一家很难快速应对。另外对那些采用了 Java 平台的企业来说,一个很现实的担忧是 Java 平台的供应商锁定(Vendor lock-in)问题。如果自己公司的核心业务系统都基于 Java 平台来构建,而这个平台本身却是另外一家公司的私有技术,有这样的顾虑情有可原。Sun 公司也意识到这一点,于是开始了 Java 平台的开放化进程。

首先开放的是 Java 平台的规范。Sun 公司公开了 Java 语言规范和 Java 虚拟机规范。Java 语言规范详细描述了 Java 语言的语法和重要特性。Java 虚拟机规范规定了 Java 字节代码规范和执行

Java 字节代码的虚拟机的相关细节。Sun 公司依托 Java 程序社区 JCP 的流程来规范对 Java 平台所做的各种修改，同时依赖社区的力量来完善 Java 平台本身。社区对 Java 平台更新的贡献的一个典型例子是 J2SE5.0 中引入的同步实用类库 `java.util.concurrent` 包。在这之前，使用 Java 进行多线程编程只能使用最基本的几个原语，不但复杂而且容易出错。Doug Lea 领导开发了方便多线程开发的实用工具包。这个工具包最初是独立分发的，后来正式成为 Java 平台的一部分。

另一部分需要开放的是 Sun 自己的 Java 虚拟机实现、Java 类库实现和编译器等实用工具。Sun 公司于 2006 年宣布 Java 将成为开放源代码的软件。为了实践这个开放策略，Sun 公司开发的 HotSpot 虚拟机和编译器最先成为使用 GPL 协议的自由软件。接着 Java 类库中的绝大部分都按照 GPL 协议开放了源代码。而对于剩下的小部分由于版权原因无法开放源代码的类库，也在社区的努力下找到了替代的开源实现。至此，Java 语言终于有了一个自由的开放源代码的实现，即 OpenJDK。

OpenJDK 的出现正在对 Java 语言产生着积极而深远的影响。依托于社区的开放源代码模式，使 Java 平台可以依靠社区的力量快速发展。而这种模式也避免了供应商锁定的问题。值得一提的是，OpenJDK 的出现并不妨碍其他私有的 JDK 的发展。其他公司仍然可以针对不同的应用环境开发出更加适合的 JDK，这些私有的 JDK 也可以自由地使用 OpenJDK 作为其实现的基础。

目前的 OpenJDK 主要有 JDK7 和 JDK6 两个版本。OpenJDK7 是目前主要开发的版本，也是 Java SE7 的参考实现。OpenJDK6 则是 Java SE6 的一个开放源代码的实现，主要被用在 Fedora 等 Linux 分发平台上。

在 Oracle 公司收购了 Sun 公司之后，Oracle 公司并没有改变在 Java 平台上的开放策略，而是继续支持 OpenJDK 的发展。随后，Oracle 同 IBM 和苹果公司都建立了合作关系，共同推进 OpenJDK 的发展。

8.2 系统架构优化建议

8.2.1 系统架构调优

8.2.1.1 系统架构过程中常用到的术语

性能：Web 系统的性能受多方面因素的影响，但大多数开发人员主要关心的是响应时间和可扩展性这两方面。

- **响应时间**：Web 应用从收到请求到返回响应结果所花费的时间。而应用系统应该在可接受的时间范围内返回响应结果，否则就不能算是一个性能良好的应用系统。
- **可扩展性**：如果 Web 应用通过增加更多硬件可以使处理的请求数呈线性增长，那么该应用是可扩展的。

在同一个应用中，响应时间和可扩展性并不总是能够同时达到最好的效果。要么应用程序有可接受的响应时间但是不能处理超过一定数量的请求；要么应用程序可以处理大量请求，但是响应时间却不尽如人意，甚至非常糟糕。因此，通常情况下我们需要在这两个要素中寻求平衡点使我们的应用系统性能达到最佳状态。

扩展的方式可以分为纵向扩展和横向扩展两种。

- **纵向扩展（垂直扩展）**：为单台机器增加 CPU 或者提高单台机器 CPU 性能。
- **横向扩展（水平扩展）**：增加服务器数量

一般情况下水平扩展比垂直扩展更重要，主要是因为普通硬件商品远比需要特殊配置的硬件便宜（比如大型机）；但是增强单个应用在一个硬件商品上处理的请求数同样也是比较重要的。同时，一个应用系统在不降低响应时间的前提下，如果通过添加更多的资源能够处理更多的请求，那么这个系统表现良好。

容量规划：需要我们根据产品期望的负载量来预估所需要的硬件数量。除了整体的预估外，这通常还包括使用更少硬件时系统的性能表现情况以及单台机器下性能的测试及评估。

架构扩展：如果 Web 应用的每一层在多层架构体系中都是可扩展的，那么该应用也具有可扩展性（横向扩展）。例如，如下图所示，我们就可以通过增加额外的资源来实现应用层和数据库层的线性扩展。

负载均衡器扩展：可以通过将 DNS 指向多个 IP 以及使用 DNS 轮循查找 IP 地址的方式来实现负载均衡器的横向扩展。或者可以使用多级负载均衡，使上一级负载均衡器来分发至下一级负载均衡器，但使用多个负载均衡器的情况比较少，主要是由于 Web 容器一般可以处理几千并发请求，而使用 Nginx 或者 HAProxy 的单个负载均衡器可以处理超过 20000 的并发请求，因此单个负载均衡器完全可以代理多个 web 应用。

数据库功能扩展：数据库功能扩展是非常常用的方式，但是扩展数据库功能（比如创建存储过程或自定义函数）都会给数据层带来额外开销以及增加数据层的复杂性。

- **关系型数据库**：可以通过主从同步的模式实现扩展，主库以写入数据为主，从库只做读操作。但是，如果业务量比较大时主从同步模式所提供的扩展能力非常有限，此外，开发人员还需要通过数据库拆分技术来进一步满足业务需求。
- **NoSQL**：CAP 定理（译者注：不熟悉的读者可通过谷歌查阅）告诉我们一个应用不可能同时满足一致性、可用性、分区容错性三个要求。而 NoSQL 一般则是通过牺牲一定的一致性来获得更高可用性以及更好的分区容错性。

数据库拆分可以垂直拆分和水平拆分：

- **垂直拆分（Partitioning）**：根据领域模型概念的基础可以将数据库拆分为几个松耦合的子库。例如，拆分为消费者数据库、产品数据库。另一种垂直拆分数据库的方式是将一个实体的一部分字段拆分出来作为一个新数据库，另一部分字段作为另一个数据库。例如，将消费者数据拆分为消费者联系信息和订单数据两部分。
- **水平拆分（Sharding）**：可以根据数据的离散属性将数据进行水平分割。例如，消费者可以根据地域不同拆分为美国消费者数据库和欧洲消费者数据库。

将数据库从单个数据库使用分区或者 sharding 拆分为多个数据库是一项非常有挑战性的任务。

8.2.1.2 系统扩展的方式

可扩展系统出现性能瓶颈的原因主要在于以下方面。

中心化组件，应用中一个不可扩展的组件直接影响整个系统或者请求处理通道所能处理请求数的上限。

高延迟组件，一个高延迟的组件会影响整个系统响应时间的下限，使整个系统响应时间更长。通常解决这个问题的办法是使高延迟的组件作为后台任务线程或者使用异步队列来解决。

CPU 消耗型应用：如果一个应用的吞吐量受 CPU 的限制，那么该应用就是 CPU 消耗型应用。此类引用通过增加 CPU 计算速度即可减少响应时间。

以下这些应用场景可能属于 CPU 消耗性。

- (1) 需要计算或者处理数据而不需要做 IO 操作的应用（财务或者交易类系统）。
- (2) 非常依赖缓存而且不做任何 IO 操作的应用。
- (3) 异步（非阻塞）模型而且不需要等待外部资源的应用（被动应用或者使用 NoJs 的应用）。

在以上使用场景中已经正常运行的应用，如果在一些应用场景中写了糟糕的或者效率低下的代码来对每次请求做大量额外的计算或者循环，那么这些应用的 CPU 占用率将非常高。但是通过分析应用可以很容易发现并修改效率低的问题。

IO 消耗型应用：如果一个应用的吞吐量受 IO 或者网络操作影响而且提升 CPU 计算速度并不能减少响应时间，那么该应用即为 IO 消耗型应用。大多数应用是 IO 消耗型应用主要是由于要做增、删、改、查操作。而性能调优和对 IO 消耗型应用进行扩展也由于这些系统依赖于其他系统的下行流量变得非常困难。

以下这些应用场景可能是 IO 消耗性。

- (1) 依赖数据库并进行增、删、改、查操作的应用。
- (2) 需要下行流量来完成本身操作的应用。

我们经常在书本上看到，系统架构好坏是评判架构师的一条准则，我并不认为存在好与坏之说，只能说存在是否适合的说法。我总结了一些常用的判断标准，分享给大家。

- 无论你怎么设计系统，系统整体化来说一定要能容易地进行水平扩展。具体一点来说，你的整个数据流过程中的所有环节都要能够做到水平扩展。这样，当你的系统存在性能问题时，“加 30 台的服务器”这样的提议才不会被人讥笑。
- 所有的技术都不是一朝一夕能搞定的，没有长期的积累，基本无望。我们可以看到，无论你用哪种都会引发一些复杂性，设计总是在做一种权衡。
- 本书第 1 章提到的 12306 票务系统，集中式的售票系统设计很难搞定，我们通过各种技术提升可以让订票系统有几百倍的性能提升。但是，总的来说，业务逻辑上的处理是能让现有系统性能有质的提升的最好方法，例如让集中式售票系统变成各点分散售票（这个不利于业务提升，基本不适用于 12306 和我国国情）。

8.2.2 Java 项目优化方式分享

一般来说，针对一个项目的优化，我们需要从多个角度分析导致性能低的原因，并逐个进行优化，最终使得程序的性能得到极大提升，增强了代码的可读性、可扩展性。

衡量一个程序是否优质，可以从多个角度进行分析。其中，最常见的衡量标准是程序的时间复杂度、空间复杂度，以及代码的可读性、可扩展性。针对程序的时间复杂度和空间复杂度，想要优化程序代码，需要对数据结构与算法有深入的理解，并且熟悉计算机系统的基本概念和原理；而针对代码的可读性和可扩展性，想要优化程序代码，需要深入理解软件架构设计，熟知并会应用合适的设计模式。

首先，如今计算机系统的存储空间已经足够大了，达到了TB级别，因此相比于空间复杂度，时间复杂度是程序员首要考虑的因素。为了追求高性能，在某些频繁操作执行时，甚至可以考虑用空间换取时间。

其次，由于受到处理器制造工艺的物理限制、成本限制，CPU主频的增长遇到了瓶颈，摩尔定律已渐渐失效，每隔18个月CPU主频即翻倍的时代已经过去了，程序员的编程方式发生了彻底的改变。在目前这个多核多处理器的时代，涌现了原生支持多线程的语言（如Java）以及分布式并行计算框架（如Hadoop）。为了使程序充分地利用多核CPU，简单地实现一个单线程的程序是远远不够的，程序员需要能够编写出并发或者并行的多线程程序。

最后，大型软件系统的代码行数达到了百万级，如果没有一个设计良好的软件架构，想在已有代码的基础上进行开发，开发代价和维护成本是无法想象的。一个设计良好的软件应该具有可读性和可扩展性，遵循“开闭原则”、“依赖倒置原则”、“面向接口编程”等。

8.2.2.1 一般性软件项目优化案例

假设我们有这么一个项目，外部系统D通过系统对外提供的REST API接口从系统内部获取信息，从中提取出有效的信息，并通过JDBC存储到某数据库系统S中，以便供系统其他部分使用，上述操作的执行频率为每天一次，一般在午夜当系统空闲时定时执行。为了实现高可用性(High Availability)，外部系统D部署在两台服务器上，因此需要分别从这两台服务器上获取信息并将信息插入数据库中，有效信息的条数达到了上千条，数据库插入操作次数则为有效信息条数的两倍。系统架构图如图8-1所示。

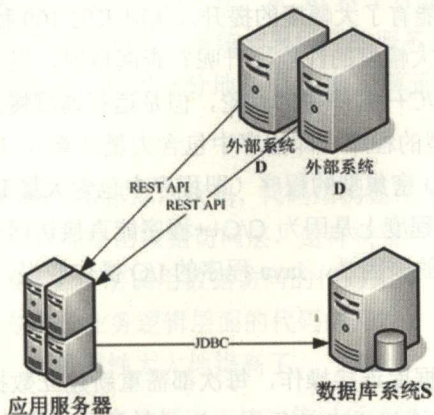


图8-1 系统架构图

为了快速实现预期效果，在最初的实现中优先考虑了功能的实现，而未考虑系统性能和代码可读性等。系统大致有以下的实现。

(1) REST API 获取信息、数据库操作可能抛出的异常信息都被记录到日志文件中，作为调

试用。

- (2) 共有 5 次数据库连接操作, 包括第一次清空数据库表, 针对两个外部系统 D 各有两次数据库插入操作, 这 5 个连接都是独立的, 用完之后即释放。
- (3) 所有的数据库插入语句都是使用 `java.sql.Statement` 类生成的。
- (4) 所有的数据库插入语句, 都是单条执行的, 即生成一条执行一条。
- (5) 整个过程都是在单个线程中执行的, 包括数据库表清空操作, 数据库插入操作, 释放数据库连接。
- (6) 数据库插入操作的 JDBC 代码散布在代码中。虽然这个版本的系统可以正常运行, 达到了预期的效果, 但是效率很低, 从通过 REST API 获取信息, 到解析并提取有效信息, 再到数据库插入操作, 总共耗时 100 秒左右。而预期的时间应该在一分钟以内, 这显然是不符合要求的。

开始分析整个过程有哪些耗时操作, 以及如何提升效率, 缩短程序执行的时间。通过 REST API 获取信息, 因为是使用外部系统提供的 API, 所以无法在此处提升效率; 取得信息之后解析出有效部分, 因为是对特定格式的信息进行解析, 所以也无效率提升的空间。综上所述, 效率可以大幅度提升的空间在数据库操作部分以及程序控制部分。

针对日志的优化

因为从两台服务器的外部系统 D 上获取到的信息是相同的, 所以数据库插入操作会抛出异常, 异常信息类似于 “Attempt to insert duplicate record”, 这样的异常信息跟有效信息的条数相等, 有上千条。这种情况是能预料到的, 所以可以考虑关闭日志记录, 或者不关闭日志记录而是更改日志输出级别, 只记录严重级别 (severe level) 的错误信息, 并将此类操作的日志级别调整为警告级别 (warning level), 这样就不会记录以上异常信息了。本项目使用的是 Java 自带的日志记录类, 以下配置文件将日志输出级别设置为严重级别。

通过上述的优化之后, 性能有了大幅度的提升, 从原来的 100 秒左右降到了 50 秒左右。为什么仅仅不记录日志就能有如此大幅度的性能提升呢? 查阅资料, 发现已经有人做了相关的研究与实验。经常听到 Java 程序比 C/C++ 程序慢的言论, 但是运行速度慢的真正原因是什么, 估计很多人并不清楚。对于 CPU 密集型的程序 (即程序中包含大量计算), Java 程序可以达到 C/C++ 程序同等级别的速度, 但是对于 I/O 密集型的程序 (即程序中包含大量 I/O 操作), Java 程序的速度就远远慢于 C/C++ 程序了, 很大程度上是因为 C/C++ 程序能直接访问底层的存储设备。因此, 不记录日志而得到大幅度性能提升的原因是, Java 程序的 I/O 操作较慢, 是一个很耗时的操作。

针对数据库连接的优化

假设程序中共有若干次数据库连接操作, 每次都需重新建立数据库连接, 数据库插入操作完成之后又立即释放了, 数据库连接没有被复用。为了做到共享数据库连接, 可以通过单例模式 (Singleton Pattern) 获得一个相同的数据库连接, 每次数据库连接操作都共享这个数据库连接。这里没有使用数据库连接池 (Database Connection Pool) 是因为在程序只有少量的数据库连接操作, 只有在大量并发数据库连接的时候才需要连接池。

通过上述的优化之后, 性能有了小幅度的提升, 从 50 秒左右降到了 40 秒左右。共享数据库

连接而得到的性能提升的原因是，数据库连接是一个耗时耗资源的操作，需要同远程计算机进行网络通信，建立 TCP 连接，还需要维护连接状态表，建立数据缓冲区。如果共享数据库连接，则只需要进行一次数据库连接操作，省去了多次重新建立数据库连接的时间。

针对数据库插入数据的优化

这个假想项目内部存在大量的数据库插入操作，所以使用预编译 SQL 方式。具体做法是使用 `java.sql.PreparedStatement` 代替 `java.sql.Statement` 生成 SQL 语句。`PreparedStatement` 使得数据库预先编译好 SQL 语句，并用来传入参数。而 `Statement` 生成的 SQL 语句在每次提交时，数据库都需进行编译。在执行大量类似的 SQL 语句时，可以使用 `PreparedStatement` 提高执行效率。使用 `PreparedStatement` 的另一个好处是不需要拼接 SQL 语句，代码的可读性更强。通过上述的优化之后，性能有了小幅度的提升，从 40 秒左右降到了 30~35 秒左右。

此外，我们可以使用 SQL 批处理方式加快 SQL 执行。通过 `java.sql.PreparedStatement` 的 `addBatch` 方法将 SQL 语句加入到批处理，这样在调用 `execute` 方法时，就会一次性地执行 SQL 批处理，而不是逐条执行。通过上述的优化之后，性能有了小幅度的提升，从 30~35 秒左右降到了 30 秒左右。

针对多线程的优化

清空数据库表的操作，把从两个外部系统 D 取得的数据插入数据库记录的操作，是相互独立的任务，可以给每个任务分配一个线程执行。清空数据库表的操作应该先于数据库插入操作完成，可以通过 `java.lang.Thread` 类的 `join` 方法控制线程执行的先后次序。在单核 CPU 时代，操作系统中某一时刻只有一个线程在运行，通过进程/线程调度，给每个线程分配一小段执行的时间片，可以实现多个进程/线程的并发（concurrent）执行。而在目前的多核多处理器背景下，操作系统中同一时刻可以有多个线程并行（parallel）执行，大大地提高了计算速度。

此外，通过采用不同的锁机制，也能加快程序的执行，具体请见第 5 章。

通过上述的优化之后，性能有了大幅度的提升，从 30 秒左右降到了 15 秒以下，10~15 秒之间。使用多线程而得到的性能提升的原因是，系统部署所在的服务器是多核多处理器的，使用多线程，给每个任务分配一个线程执行，可以充分地利用 CPU 计算资源。

针对设计模式的优化

原来的代码中混杂着 JDBC 操作数据库的代码，代码结构显得十分凌乱。通过使用 DAO 模式（Data Access Object Pattern）可以抽象出数据访问层，这样使得程序可以独立于不同的数据库，即便访问数据库的代码发生了改变，上层调用数据访问的代码无须改变。并且程序员可以摆脱单调烦琐的数据库代码的编写，专注于业务逻辑层面的代码的开发。通过上述的优化之后，性能并未有提升，但是代码的可读性、可扩展性大大地提高了。

总的来说，通过关闭日志记录、共享数据库连接、使用预编译 SQL、使用 SQL 批处理、使用多线程实现并发/并行、使用 DAO 模式抽象出数据访问层，程序运行时间较大幅度地被缩短了，在性能上得到了很大的提升，同时也具有了更好的可读性和可扩展性。

8.2.2.2 订单系统的优化方案

要优化订单系统，提高订单的每秒交易数量（TPS，Transaction per second），我们首先要做的

是对下订单的逻辑进行剥离，只保留核心部分，而把附加功能剔除出去。比如说下单要考虑库存量、考虑发短信、要给卖家发消息通知、要对订单做统计、要做销售额统计等，我们需要对这些功能进行分析，哪些功能是必需的，哪些是附加的功能，要最大程度提高下单这一步的 TPS，就要先不考虑这些附加的功能。

下单必然会涉及到买家查看订单，和卖家查看收到的订单，修改订单价格等，这是下订单的核心。在下单这个操作中有买家和卖家两个密切相关而有不同的视角，可以称为两个不同的维度。一般来说，订单系统的下单过程这一步涉及到少数几张核心数据表，而这几张数据表涵盖了这两个维度的操作。

下单是在一个数据库事务中进行的，要提高数据库的事务并发数，最有效的办法是拆分，拆分有两种，一是对库进行拆分，另一种是在同一个库中对表进行拆分。要做拆分首先就要考虑拆分依据的字段，淘宝是根据订单号做拆分的，而下单中有两个维度，买家和卖家，对订单做拆分之后，必须还是可以通过买家，卖家方便地查询着两个维度的数据。该怎么办呢？假设我们需要拆分的数据库的规模，订单表拆分到 16 个 mysql 库中，而在每个库中又将订单表横向拆分为 64 份，相当于将一个表拆分为 1024 份。拆分之后事务会分散到 1024 套表中，这必然会很大程度上增加并发的处理事务能力。上面留了一个疑问，经过拆分之后如何保证买家卖家快速的查询其下的订单呢？最好的办法是保证买家，卖家下的订单在一张表中，如何保证呢？淘宝的做法是将买家的 id 取模后放到订单号中。假定一个订单号是 142424594267664；这个订单号对应的订单该放在哪台服务器上的哪个表中，是根据订单的后四位 7667，对 1024 取模之后决定的；同时 7667 是买家 id 的后四位。这样买家在查询其订单时就可以通过其 id 获得其订单所在库以及表，就可以方便有效的查询买家订单了。这里会带来另外一个问题，卖家查询订单时怎么办？前面我们已经提到卖家和买家被分成两个不同的维度来做表设计，卖家查询时不是直接查订单表，而是通过卖家维度的表来做查询。卖家维度的表的插入，更新是通过在订单插入时发一个消息来通知插入的。同样对于发短信、发旺旺（淘宝专有）也是通过消息来处理的，这些附加功能不参与到下单的事务中去。

即使这样做了库、表的拆分，依然会有问题。淘宝在双 11 时的一天的交易量就达到了 5000 多万，这样几个月过去后，这些拆分后的表中的数据量也会达到很大的一个量，处理速度就会下降。淘宝的做法是把三个月之前的老数据迁移到其他库中，这样就避免了数据量增大导致的系统响应时间降低的问题。但是会带来另外一个问题，用户在查询订单时需要同时查两个库，一个是历史数据表，另一个是近期数据表；这个问题无可避免，就是通过查询两次解决。

也许有的朋友会想到拆分之后对全数据做统计会有问题。如果在拆分后的表上做统计，是肯定会有问题的。怎么做呢？很简单，把数据迁移到别的库中去做统计。

表做拆分可以大大地提高 TPS，但是也会带来一些问题，需要通过可靠的消息通知机制通知其他模块做非核心处理的事情，需要通过高效的搜索系统保证搜索数据的及时更新。

8.2.3 面向服务架构

面向服务架构的思想在整个软件的架构中已经不是什么新鲜的东西。我简单地认为服务化是模块化的延伸，所以服务化有着和模块化类似的优点和缺点。无论你采用哪种协议定义服务与服务之间的通信方式（如 WebServices、私有协议等），这并不是服务化的本质所在，即使 Java 语言

用 RMI 进行服务与服务之间的通信也仍然不违背服务化的宗旨。

为什么需要面向服务架构

- 我觉得面向服务的根本好处是便于管理，也是应用大到一定时候的必然产物。这往往和组织架构之间相契合。其实不合理的服务划分也会带来服务之间的混乱。
- 面向服务是一个解耦的过程，松耦合降低了服务之间的依赖，也意味着服务一个服务出现故障的时候不容易引起连锁反应，也能更好的控制服务与服务之间的关系与优先级。
- 不同语言之间的通信。

8.2.3.1 SOA

面向服务架构，它可以根据需求通过网络对松散耦合的粗粒度应用组件进行分布式部署、组合和使用。服务层是 SOA 的基础，可以直接被应用调用，从而有效控制系统中与软件代理交互的人为依赖性。SOA 是一种粗粒度、松耦合服务架构，服务之间通过简单、精确定义接口进行通讯，不涉及底层编程接口和通讯模型。SOA 可以看作是 B/S 模型、XML（标准通用标记语言的子集）/Web Service 技术之后的自然延伸。SOA 将能够帮助软件工程师们站在一个新的高度理解企业级架构中的各种组件的开发、部署形式，它将帮助企业系统架构者以更迅速、更可靠、更具重用性架构整个业务系统。较之以往，以 SOA 架构的系统能够更加从容地面对业务的急剧变化。

传统的 Web（HTML/HTTP）技术有效地解决了人与信息系统的交互和沟通问题，极大的促进了 B2C 模式的发展。WEB 服务(XML/SOAP/WSDL) 技术则是要有效的解决信息系统之间的交互和沟通问题，促进 B2B/EAI/CB2C 的发展。SOA（面向服务的体系）则是采用面向服务的商业建模技术和 WEB 服务技术，实现系统之间的松耦合，实现系统之间的整合与协同。WEB 服务和 SOA 的本质思路在于使得信息系统个体在能够沟通的基础上形成协同工作。

对于面向同步和异步应用的，基于请求/响应模式的分布式计算来说，SOA 是一场革命。一个应用程序的业务逻辑（business logic）或某些单独的功能被模块化并作为服务呈现给消费者或客户端。这些服务的关键是他们的松耦合特性。例如，服务的接口和实现相独立。应用开发人员或者系统集成者可以通过组合一个或多个服务来构建应用，而无须理解服务的底层实现。举例来说，一个服务可以用 .NET 或 J2EE 来实现，而使用该服务的应用程序可以在不同的平台之上，使用的语言也可以不同。

SOA 具有的特性

SOA 服务具有平台独立的自我描述 XML 文档。Web 服务描述语言（WSDL, Web Services Description Language）是用于描述服务的标准语言。

SOA 服务用消息进行通信，该消息通常使用 XML Schema 来定义（也叫作 XSD, XML Schema Definition）。消费者和提供者或消费者和服务之间的通信多见于不知道提供者的环境中。服务间的通讯也可以看作企业内部处理的关键商业文档。

在一个企业内部，SOA 服务通过一个扮演目录列表（directory listing）角色的登记处（Registry）来进行维护。应用程序在登记处（Registry）寻找并调用某项服务。统一描述，定义和集成（UDDI, Universal Description, Definition, and Integration）是服务登记的标准。

每项 SOA 服务都有一个与之相关的服务品质（QoS, quality of service）。QoS 的一些关键元

素有安全需求(例如认证和授权),可靠通信(可靠消息是指,确保消息仅发送一次,从而过滤重复信息。),以及谁能调用服务的策略。

SOA 三大基本特征

1) 独立的功能实体

在 Internet 这样松散的使用环境中,任何访问请求都有可能出错,因此任何企图通过 Internet 进行控制的结构都会面临严重的稳定性问题。SOA 非常强调架构中提供服务的功能实体的完全独立自主的能力。传统的组件技术,如.NET Remoting, EJB, COM 或者 CORBA,都需要有一个宿主(Host 或者 Server)来存放和管理这些功能实体;当这些宿主运行结束时这些组件的寿命也随之结束。这样当宿主本身或者其他功能部分出现问题的时候,在该宿主上运行的其他应用服务就会受到影响。

SOA 架构中非常强调实体自我管理和恢复能力。常见的用来进行自我恢复的技术,比如事务处理(Transaction),消息队列(Message Queue),冗余部署(Redundant Deployment)和集群系统(Cluster)在 SOA 中都起到至关重要的作用。

2) 大数据量低频率访问

对于.NET Remoting, EJB 或者 XML-RPC 这些传统的分布式计算模型而言,他们的服务提供都是通过函数调用的方式进行的,一个功能的完成往往需要通过客户端和服务端来回很多次数函数调用才能完成。在 Intranet 的环境下,这些调用给系统的响应速度和稳定性带来的影响都可以忽略不计,但是在 Internet 环境下这些因素往往是决定整个系统是否能正常工作的一个关键决定因素。因此 SOA 系统推荐采用大数据量的方式一次性进行信息交换。

3) 基于文本的消息传递

由于 Internet 中大量异构系统的存在决定了 SOA 系统必须采用基于文本而非二进制的消息传递方式。在 COM、CORBA 这些传统的组件模型中,从服务器端传往客户端的是一个二进制编码的对象,在客户端通过调用这个方法来完成某些功能;但是在 Internet 环境下,不同语言,不同平台对数据、甚至是一些基本数据类型定义不同,给不同的服务之间传递对象带来的很大困难。由于基于文本的消息本身是不包含任何处理逻辑和数据类型的,因此服务间只传递文本,对数据的处理依赖于接收端的方式可以帮忙绕过兼容性这个大泥坑。

此外,对于一个服务来说,Internet 与局域网最大的一个区别就是在 Internet 上的版本管理极其困难,传统软件采用的升级方式在这种松散的分布式环境中几乎无法进行。采用基于文本的消息传递方式,数据处理端可以只选择性的处理自己理解的那部分数据,而忽略其他的数据,从而得到的非常理想的兼容性。

8.2.3.2 微服务架构 (Microservices)

对微服务架构我们没有一个明确的定义,但简单来说微服务架构是采用一组服务的方式来构建一个应用,服务独立部署在不同的进程中,不同服务通过一些轻量级交互机制来通信,例如 RPC、HTTP 等,服务可独立扩展伸缩,每个服务定义了明确的边界,不同的服务甚至可以采用不同的编程语言来实现,由独立的团队来维护。

微服务架构特征 (Characteristics) 包括如下这些特点:

1) 通过服务实现组件化

传统实现组件的方式是通过库 (library)，传统组件是和应用一起运行在进程中，组件的局部变化意味着整个应用的重新部署。通过服务来实现组件，意味着将应用拆散为一系列的服务运行在不同的进程中，那么单一服务的局部变化只需重新部署对应的服务进程。另外将服务作为组件可以更明确地定义出组件的边界，因为服务之间的调用是跨进程的，清晰的边界和职责定义是设计时必须考虑的。

2) 按业务能力来划分服务与组织团队

康威定律 (Conway's law) 指出，任何设计系统的组织，最终产生的设计等同于组织之内、之间的沟通结构。

传统开发方式中，我们将工程师按技能专长分层为前端层、中间层、数据层，前端对应的角色为 UI、页面构建师等，中间层对应的角色为服务端业务开发工程师，数据层对应着 DBA 等角色。事实上传统应用设计架构的分层结构正反映了不同角色的沟通结构。而微服务架构的开发模式不同于传统方式，它将应用按业务能力来划分为不同的服务，每个服务都要求在对应业务领域的全栈（从前端到后端）软件实现，从界面到数据存储到外部沟通协作等。因此团队的组织是跨功能的，包含实现业务所需的全面的技能。近年兴起的全栈工程师正是因为架构和开发模式的转变而出现，当然具备全栈的工程师其实很少，但将不同领域的工程师组织为一个全栈的团队就容易得多。

3) 服务即产品

传统的应用开发都是基于项目模式的，开发团队根据一堆功能列表开发出一个软件应用并交付给客户后，该软件应用就进入维护模式，由另一个维护团队负责，开发团队的职责结束。而微服务架构的倡导者提议避免采用这种项目模式，更倾向于让开发团队负责整个产品的全部生命周期。Amazon 对此提出了一个观点：

开发团队对软件在生产环境的运行负全部责任，让服务的开发者与服务的使用者（客户）形成每天交流反馈，来自直接客户端的反馈有助于开发者提升服务的质量。

4) 智能终端与哑管道

微服务架构抛弃了 ESB 过度复杂的业务规则编排、消息路由等。服务作为智能终端，所有的业务智能逻辑在服务内部处理，而服务间的通信尽可能的轻量化，不添加任何额外的业务规则。

5) 去中心统一化

传统应用中倾向采用统一的技术平台或产品来解决所有问题。不是每个问题都是钉子，也不是每个解决方案都是一个锤子。问题有其具体性，解决方案也应有其针对性。用最适合的技术方案去解决具体的问题，在大一统的传统应用中其实很难做到，而微服务的架构意味着，你可以针对不同的业务服务特征选择不同的技术平台或产品，有针对性的解决具体的业务问题。

6) 基础设施自动化

单一进程的传统应用被拆分为一系列的多进程服务后，意味着开发、调试、测试、集成、监控和发布的复杂度都会相应增大。必须要有合适的自动化基础设施来支持微服务架构模式，否则开发、运维成本将大大增加。

7) Design for failure

正因为将服务独立在不同的进程中后，引入了额外的失败因素。任何时刻对服务的调用都可能因为服务方不可用导致失败，这就要求服务的消费方需要优雅的处理此类错误。这其实是相对传统应用开发方式的一个缺点，不过随着一些开源服务化框架的出现，对业务开发人员而言适当的屏蔽了类似的错误处理，不过开发人员依然需要知道对服务的调用是完全不同于进程内的方法或函数调用的。

8) 进化设计

一旦采用了微服务架构模式，那么在服务需要变更时我们要特别小心，服务提供者的变更可能引发服务消费者的兼容性破坏，时刻谨记保持服务契约（接口）的兼容性。对于解耦服务消费方和服务提供方，伯斯塔尔法则（Postel's law）特别适用，即发送时要保守，接收时要开放。

按照伯斯塔尔法则的思想来设计实现服务调用时，发送的数据要更保守，意味着最小化的传送必要的信息，接收时更开放意味着要最大限度的容忍信息的兼容性。多余的信息不认识可以忽略，而不应该拒绝或抛出错误。

微服务架构应用

采用微服务架构面临的第一个问题就是如何将一个单一应用拆分为多个服务。有一个一般的原则是，单一服务提供的功能是可以独立被替换和升级的。也就是说如果有 A 和 B 两个功能，如果 A 功能发生变化时同时 B 功能也需要变化，那么 A 和 B 这两个功能应该被划在一个服务中。

微服务架构应用的成功经验近年已越来越多，例如国外的 Amazon, Netflix，国内如阿里都采用微服务架构取得了很多正面的成功案例。但通过上文所述微服务架构特征看出，其实微服务架构模式有利有弊，需要根据实际的业务、团队、环境进行仔细权衡利弊。其中的服务拆分带来的额外开发、测试、运维、监控的复杂度，在现有的环境、团队下是否能够很好的支持。

另外，有人可能会说，我一开始不采用微服务架构方式，而是在单一进程内基于清晰定义的模块化方式，模块之间通过接口调用，到了适当阶段，必要的时候再将模块拆分为服务。其实这个想法显得过于理想，因为进程内良好定义的接口通常不是很好的服务化接口。一开始没有考虑服务化的设计方法，那么后期拆分时依然是一个痛苦的过程。

总的来说，面向服务架构是一种思想，当然对于大系统而言其利必大于弊，而系统比较小的时候盲目的拆分和服务化其实会导致整个维护成本上升。系统架构并没有一成不变的套路，也不必完全遵循某种模式。一切都在实际应用中结合具体的应用场景。应该说是“方法论”的具体产物。

8.2.4 程序隔离技术

CGroup 是 Control Groups 的缩写，是 Linux 内核提供了一种可以限制、记录、隔离进程组（process groups）所使用的物力资源（如 cpu memory i/o 等）的机制。2007 年进入 Linux 2.6.24 内核，CGROUPs 不是全新创造的，它将进程管理从 cpuset 中剥离出来，作者是 Google 的 Paul Menage。CGROUPs 也是 LXC 为实现虚拟化所使用的资源管理手段。

CGroup 是将任意进程进行分组化管理的 Linux 内核功能。CGroup 本身是提供将进程进行分

组化管理的功能和接口的基础结构，I/O 或内存的分配控制等具体的资源管理功能是通过这个功能来实现的。这些具体的资源管理功能称为 CGroup 子系统或控制器。CGroup 子系统有控制内存的 Memory 控制器、控制进程调度的 CPU 控制器等。运行中的内核可以使用的 Cgroup 子系统由 /proc/cgroup 来确认。

CGroup 提供了一个 CGroup 虚拟文件系统，作为进行分组管理和各子系统设置的用户接口。要使用 CGroup，必须挂载 CGroup 文件系统。这时通过挂载选项指定使用哪个子系统。

CGroup 相关概念解释如下所示。

- (1) 任务 (task)。在 cgroups 中，任务就是系统的一个进程。
- (2) 控制族群 (control group)。控制族群就是一组按照某种标准划分的进程。Cgroups 中的资源控制都是以控制族群为单位实现。一个进程可以加入到某个控制族群，也从一个进程组迁移到另一个控制族群。一个进程组的进程可以使用 cgroups 以控制族群为单位分配的资源，同时受到 cgroups 以控制族群为单位设定的限制。
- (3) 层级 (hierarchy)。控制族群可以组织成 hierarchical 的形式，既一颗控制族群树。控制族群树上的子节点控制族群是父节点控制族群的孩子，继承父控制族群的特定的属性。
- (4) 子系统 (subsystem)。一个子系统就是一个资源控制器，比如 cpu 子系统就是控制 cpu 时间分配的一个控制器。子系统必须附加 (attach) 到一个层级上才能起作用，一个子系统附加到某个层级以后，这个层级上的所有控制族群都受到这个子系统的控制。

每次在系统中创建新层级时，该系统中的所有任务都是那个层级的默认 CGroup (我们称之为 Root CGroup，此 CGroup 在创建层级时自动创建，后面在该层级中创建的 CGroup 都是此 CGroup 的后代) 的初始成员，一个子系统最多只能附加到一个层级，一个层级可以附加多个子系统，一个任务可以是多个 CGroup 的成员，但是这些 CGroup 必须在不同的层级。系统中的进程 (任务) 创建子进程 (任务) 时，该子任务自动成为其父进程所在 CGroup 的成员。然后可根据需要将该子任务移动到不同的 CGroup 中，但开始时它总是继承其父任务的 CGroup。

如图 8-2 所示的 CGroup 层级关系显示，CPU 和 Memory 两个子系统有自己独立的层级系统，而又通过 Task Group 取得关联关系。

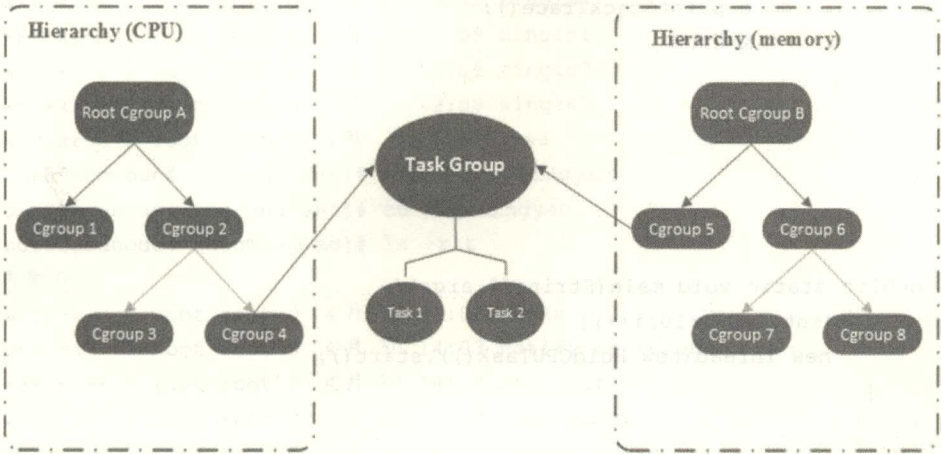


图8-2 CGroup层级图

实验基于 Linux Centosv6.564 位版本, JDK1.7。实验目的是运行一个占用 CPU 的 Java 程序, 如果不用 CGroup 物理隔离 CPU 核, 那程序会由操作系统层级自动挑选 CPU 核来运行程序。由于操作系统层面采用的是时间片轮询方式随机挑选 CPU 核作为运行容器, 所以会在本机器上 24 个 CPU 核上随机执行。如果采用 CGroup 进行物理隔离, 我们可以选择某些 CPU 核作为指定运行载体。

代码清单 8-2 创建一个占据 CPU 的任务

// 开启 4 个用户线程, 其中 1 个线程大量占用 CPU 资源, 其他 3 个线程则处于空闲状态

```
public class HoldCPUMain {
    public static class HoldCPUTask implements Runnable{
```

```
        @Override
```

```
        public void run() {
```

```
            // TODO Auto-generated method stub
```

```
            while(true){
```

```
                double a = Math.random()*Math.random();//占用 CPU
```

```
                System.out.println(a);
```

```
            }
```

```
        }
```

```
    public static class LazyTask implements Runnable{
```

```
        @Override
```

```
        public void run() {
```

```
            // TODO Auto-generated method stub
```

```
            while(true){
```

```
                try {
```

```
                    Thread.sleep(1000);
```

```
                } catch (InterruptedException e) {
```

```
                    // TODO Auto-generated catch block
```

```
                    e.printStackTrace();
```

```
                } //空闲线程
```

```
            }
```

```
        }
```

```
    }
```

```
    public static void main(String[] args){
```

```
        for(int i=0;i<10;i++){
```

```
            new Thread(new HoldCPUTask()).start();
```

```
        }
```

```
    }
```

```
}
```


清单 8-1 所示程序会启动 10 个线程，这 10 个线程都在做占用 CPU 的计算工作，它们可能会运行在 1 个 CPU 核上，也可能运行在多个核上，由操作系统决定。我们稍后会在 Linux 机器上通过命令在后台运行清单 1 程序。本实验需要对 CPU 资源进行限制，所以我们在 `cpu_and_set` 子系统上创建自己的层级“zhoumingyao”。

代码清单 8-3 创建自己的层级

```
[root@facenode4 cpu_and_set]# ls -rlt
总用量 0
-rw-r--r-- 1 root root 0 3月 21 17:21 release_agent
-rw-r--r-- 1 root root 0 3月 21 17:21 notify_on_release
-r--r--r-- 1 root root 0 3月 21 17:21 cpu.stat
-rw-r--r-- 1 root root 0 3月 21 17:21 cpu.shares
-rw-r--r-- 1 root root 0 3月 21 17:21 cpuset.sched_relax_domain_level
-rw-r--r-- 1 root root 0 3月 21 17:21 cpuset.sched_load_balance
-rw-r--r-- 1 root root 0 3月 21 17:21 cpuset.mems
-rw-r--r-- 1 root root 0 3月 21 17:21 cpuset.memory_spread_slab
-rw-r--r-- 1 root root 0 3月 21 17:21 cpuset.memory_spread_page
-rw-r--r-- 1 root root 0 3月 21 17:21 cpuset.memory_pressure_enabled
-r--r--r-- 1 root root 0 3月 21 17:21 cpuset.memory_pressure
-rw-r--r-- 1 root root 0 3月 21 17:21 cpuset.memory_migrate
-rw-r--r-- 1 root root 0 3月 21 17:21 cpuset.mem_hardwall
-rw-r--r-- 1 root root 0 3月 21 17:21 cpuset.mem_exclusive
-rw-r--r-- 1 root root 0 3月 21 17:21 cpuset.cpus
-rw-r--r-- 1 root root 0 3月 21 17:21 cpuset.cpu_exclusive
-rw-r--r-- 1 root root 0 3月 21 17:21 cpu.rt_runtime_us
-rw-r--r-- 1 root root 0 3月 21 17:21 cpu.rt_period_us
-rw-r--r-- 1 root root 0 3月 21 17:21 cpu.cfs_quota_us
-rw-r--r-- 1 root root 0 3月 21 17:21 cpu.cfs_period_us
-r--r--r-- 1 root root 0 3月 21 17:21 cgroup.procs
drwxr-xr-x 2 root root 0 3月 21 17:22 test
drwxr-xr-x 2 root root 0 3月 23 16:36 test1
-rw-r--r-- 1 root root 0 3月 25 19:23 tasks
drwxr-xr-x 2 root root 0 3月 31 19:32 single
drwxr-xr-x 2 root root 0 3月 31 19:59 single1
drwxr-xr-x 2 root root 0 3月 31 19:59 single2
drwxr-xr-x 2 root root 0 3月 31 19:59 single3
drwxr-xr-x 3 root root 0 4月 3 17:34 aaaa

[root@facenode4 cpu_and_set]# mkdir zhoumingyao
[root@facenode4 cpu_and_set]# cd zhoumingyao
[root@facenode4 zhoumingyao]# ls -rlt
总用量 0
-rw-r--r-- 1 root root 0 4月 30 14:03 tasks
-rw-r--r-- 1 root root 0 4月 30 14:03 notify_on_release
-r--r--r-- 1 root root 0 4月 30 14:03 cpu.stat
-rw-r--r-- 1 root root 0 4月 30 14:03 cpu.shares
-rw-r--r-- 1 root root 0 4月 30 14:03 cpuset.sched_relax_domain_level
-rw-r--r-- 1 root root 0 4月 30 14:03 cpuset.sched_load_balance
```

```

-rw-r--r-- 1 root root 0 4月 30 14:03 cpuset.mems
-rw-r--r-- 1 root root 0 4月 30 14:03 cpuset.memory_spread_slab
-rw-r--r-- 1 root root 0 4月 30 14:03 cpuset.memory_spread_page
-r--r--r-- 1 root root 0 4月 30 14:03 cpuset.memory_pressure
-rw-r--r-- 1 root root 0 4月 30 14:03 cpuset.memory_migrate
-rw-r--r-- 1 root root 0 4月 30 14:03 cpuset.mem_hardwall
-rw-r--r-- 1 root root 0 4月 30 14:03 cpuset.mem_exclusive
-rw-r--r-- 1 root root 0 4月 30 14:03 cpuset.cpus
-rw-r--r-- 1 root root 0 4月 30 14:03 cpuset.cpu_exclusive
-rw-r--r-- 1 root root 0 4月 30 14:03 cpu.rt_runtime_us
-rw-r--r-- 1 root root 0 4月 30 14:03 cpu.rt_period_us
-rw-r--r-- 1 root root 0 4月 30 14:03 cpu.cfs_quota_us
-rw-r--r-- 1 root root 0 4月 30 14:03 cpu.cfs_period_us
-r--r--r-- 1 root root 0 4月 30 14:03 cgroup.procs

```

通过 `mkdir` 命令新建文件夹 `zhoumingyao`，由于已经预先加载 `cpu_and_set` 子系统成功，所以当文件夹创建完毕的同时，`cpu_and_set` 子系统对应的文件夹也会自动创建。运行 Java 程序前，我们需要确认 `cpu_and_set` 子系统安装的目录，如清单 8-4 所示。

代码清单 8-4 确认目录

```

[root@facenode4 zhoumingyao]# lscgroup
cpuacct:/
devices:/
freezer:/
net_cls:/
blkio:/
memory:/
memory:/test2
cpuset,cpu:/
cpuset,cpu:/zhoumingyao
cpuset,cpu:/aaaa
cpuset,cpu:/aaaa/bbbb
cpuset,cpu:/single3
cpuset,cpu:/single2
cpuset,cpu:/single1
cpuset,cpu:/single
cpuset,cpu:/test1
cpuset,cpu:/test

```

输出显示 `cpuset_cpu` 的目录是 `cpuset,cpu:/zhoumingyao`，由于本实验所采用的 Java 程序是多线程程序，所以需要使用 `cgexec` 命令来帮助启动，而不能如网络上有些材料所述，采用 `java -jar` 命令启动后，将 `pid` 进程号填入 `tasks` 文件即可的错误方式。清单 8-4 即采用 `cgexec` 命令启动 java 程序，需要使用到清单 8-4 定位到的 `cpuset_cpu` 目录地址。

代码清单 8-5 运行 Java 程序

```

[root@facenode4 zhoumingyao]# cgexec -g cpuset,cpu:/zhoumingyao java -jar
test.jar

```


我们在 `cpuset.cpus` 文件中设置需要限制只有 0-10 这 11 个 CPU 核可以被用来运行上述清单 4 启动的 Java 多线程程序。当然 CGroup 还可以限制具体每个核的使用百分比，这里不再做过多的描述，请读者自行翻阅 CGroup 官方材料。

接下来，通过 TOP 命令获得清单 4 启动的 Java 程序的所有相关线程 ID，将这些 ID 写入到 Tasks 文件。

代码清单 8-6 设置 CPU 线程

```
[root@facenode4 zhoumingyao]# cat tasks
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2714
2715
2716
2718
```

全部设置完毕后，我们可以通过 TOP 命令查看具体的每一颗 CPU 核上的运行情况，发现只有 0-10 这 11 颗 CPU 核上有计算资源被调用，可以进一步通过 TOP 命令确认全部都是清单 8-4 所启动的 Java 多线程程序的线程。

代码清单 8-7 运行结果

```
top - 14:43:24 up 44 days, 59 min, 6 users, load average: 0.47, 0.40, 0.33
Tasks: 715 total, 1 running, 714 sleeping, 0 stopped, 0 zombie
Cpu0 : 0.7%us, 0.3%sy, 0.0%ni, 99.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu1 : 1.0%us, 0.7%sy, 0.0%ni, 98.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu2 : 0.3%us, 0.3%sy, 0.0%ni, 99.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu3 : 1.0%us, 1.6%sy, 0.0%ni, 97.5%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu4 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu5 : 1.3%us, 1.9%sy, 0.0%ni, 96.8%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu6 : 3.8%us, 5.4%sy, 0.0%ni, 90.8%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu7 : 7.7%us, 9.9%sy, 0.0%ni, 82.4%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu8 : 4.8%us, 6.1%sy, 0.0%ni, 89.1%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu9 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu10 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu11 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu12 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu13 : 0.0%us, 0.0%sy, 0.0%ni, 72.8%id, 0.0%wa, 0.0%hi, 4.3%si, 0.0%st
Cpu14 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu15 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu16 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu17 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu18 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu19 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu20 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu21 : 0.3%us, 0.3%sy, 0.0%ni, 99.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu22 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu23 : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 32829064k total, 5695012k used, 27134052k free, 533516k buffers
Swap: 24777720k total, 0k used, 24777720k free, 3326940k cached
```

总体来说，CGroup 的使用方式较为简单，目前主要的问题是网络上已有的中文材料缺少详细的配置步骤，一旦读者通过反复实验，掌握了配置方式，使用上应该不会有大的问题。

8.2.5 团队并行开发准则

我们先来谈谈什么是并行开发的风险？并行开发最大的风险就是风险扩散，本来只是一段程序的错误或异常，逐步波及一个功能，一个模块，甚至到最后毁坏了整个项目，为什么并行开发就有这个风险呢？一个团队，20 人开发，各人负责不同的功能模块，甲负责汽车类的建造，乙负责司机类的建造，在甲没有完成的情况下，乙是不能完全地编写代码的，缺少汽车类，编译器根本就不会让你通过！在缺少 Benz 类的情况下，Driver 类能编译吗？更不要说是单元测试了！在这种不使用依赖倒置原则的环境中，所有的开发工作都是“单线程”的，甲做完，乙再做，然后是丙继续…，这在 90 年代“个人英雄主义”编程模式中还是比较适用的，一个人完成所有的代码工

作,但在现在的大中型项目中已经是不能完全胜任了,一个项目是一个团队的协作结果,一个“英雄”再牛也不可能了解所有的业务和所有的技术,要协作就要并行开发,要并行开发就要解决模块之间的项目依赖关系。

假设一个有4名编程人员的小型开发团队(以下简称“TJRP 开发组”),成员 Tom、Jason、Robert 和 Pat 分别负责4个模块,按照传统的软件开发模式,开发将经历编程-连调-测试-发布4个阶段。如果最初的设计正确,并且,四个开发人员都是资深程序员而且配合默契,那么这个模式将会运转良好。然而遗憾的是,Pat 刚刚参加工作不久,对于设计师撰写的设计文档的理解不够透彻,而 Robert 则自作主张地对设计进行了一些修正,更糟糕的是,项目组会议的时候,这些问题没有及时地被暴露出来,导致 Pat 和 Robert 代码在设计上的“分歧”越来越大。结果进入联调的阶段,Pat 和 Robert 发生了激烈的争执,在吵得不可开交之后,项目经理终于让4个人坐到了一起来解决问题,最后,连调阶段整整多花了一倍的时间。

但倒霉的事情还没有结束。Jason 在测试中发现,原本正常的代码行为被改变了,并且,他惊讶地发现代码被某个“别人”改过,在翻箱倒柜地找出某份正常版本的副本之后,他又发现,“别人”修改中的某些地方是必要的。代码合并和重新测试使得测试阶段足足花掉了原先预想3倍的时间。

可怜的 Tom 运气更差,作为主要的代码复审人员,他不得不阅读所有的代码。Robert 和 Pat 的争吵导致了大量的代码变动,他不得不重新审核代码,而 Jason 的代码合并引发的新问题又让他不得不分神去帮助 Jason 进行调整。

最后的结果是,软件开发的成本是预期的2.4倍,发布时间也拖后了不少。我并不是在开玩笑,上面所讲的是一个发生在那家安全软件公司的真实故事。他们的技术经理介绍,在施行了规范的开发制度,以及启用并行版本控制系统之后,他们认为开发达到了一个全新的水平。并行版本控制系统本身并没有产生任何代码,但由于使用这样的系统,开发的效率被大大地提高了。

所谓版本控制其实并不是什么复杂的概念。对于开发活动的绝大多数参与者来说,版本控制系统在某种意义上能够帮助他们做好开发过程中的记录工作,并且,通过保存文件在不同时期的版本,交付工程师和代码复审员能够很容易地缩小搜索问题代码的范围,而程序员则可以通过这样的系统更好地并行协作。一般来说,源代码的版本控制系统能够实现以下一些最基本的功能。

- 保存任意一个源代码文件的不同版本。
- 记录修改者、修改原因。
- 当两个用户同时修改一个文件时,尽可能地自动合并修改;在不能合并时,给出提示。
- 比较不同版本之间,或与本地副本之间的差异。
- 获取最新版本的全部源代码供测试,并允许回退到所保存的源代码的任意版本。
- 创建代码分支,便于软件发布和后期维护(后面将会提到);新的代码可以合并到这些分支中。
- 对不同的源代码给出标记,方便日后审查。
- 访问控制:阻止未经授权的修改和查阅。

我们知道，技术不是解决一切问题的灵丹妙药，但是谁也不会否认大规模的机械化生产的效率高于人拉肩扛的手工业作坊，一旦运用得当，技术将极大地改善我们的工作和生活。我们可以看到，上面的功能有效地解决了 TJRP 开发组所面临的绝大部分问题，例如。

- 由于能够同时修改代码，并获取对方的修改，Pat 和 Robert 能够有效地、尽早地进行沟通。
- 促进开发者之间的交流，每一个修改都必须给出原因，并记录提交者。
- 测试和连调可以尽早开始，避免模块之间的不兼容在最后阶段被暴露出来而阻碍发布。
- 不同开发者的修改能够及时合并，并避免由于版本不一致导致的冲突。
- 代码复审可以针对某一代码分支进行，从而，允许一些开发者持续地开发下一个版本，而稳定的代码则可以交付给用户。

更进一步，以管理者的角度，还有了一些额外的好处，如。

- 每日构建和测试能够让项目经理更好地把握工程的进度。
- 谁作了多少工作，谁工作的更出色，可以在版本控制系统中清晰地体现。
- 分工明确，通过访问控制，可以避免不了解整个代码体系的开发人员偶然的错误修改导致的全盘崩溃。
- 更重要的是，版本控制系统中将保持大量的开发经验，这对于一个开发团队来说是一笔无价的财富。

我们可以看到，上述改进集中地体现了一个重要的思想，即及时沟通以预防问题的出现，尽早发现、尽早解决问题；明确奖惩制度，激发开发人员的积极性。

8.3 与编程无关

这一小节的内容与程序优化无关，但是作者个人认为，一个程序员的气质是与生俱来+梦想+坚持+机遇组成的，所以想和大家分享一下自己对于工程师品格、工程师能力拓展、工程师思维方法拓展等方面的一些自己的认知。

8.3.1 工程师品格

写到了本文的最后，作者个人感觉，再多的性能优化经验、案例，如果没有人去阅读、尝试，都不会起多大的作用，所以工程师个人的职业规划、修养、品格，还是非常重要的，只有内心充满着技术情节的工程师，才可能会重视程序的整体性能优化策略。一般来说，工程师品格包括如下几点。

- (1) 写代码、看别人代码和文档的激情，如果只是因为这份工作可以赚钱，那我觉得你的工程师道路不会太长，你的技术造诣也不会很高。
- (2) 我读书的时候，自习室那是要抢的，大家都在努力学习，本科 4 年至少也应该写 10 万行代码。现在我面试的时候，问 985 大学即将踏入社会的研究生，“你热爱编程吗？”“你会每天晚上自学，周末到公司自己加班吗？”我很少收到肯定的答复。没有人喜欢一天

到晚学习，但是想要成为技术大牛，只有不断学习，让技术成为你的爱好，没有捷径。

- (3) 天赋，就是所谓的聪明。个人认为，好的程序员通常可能是你认识的人里最聪明的那个，而且出乎意料的，好的程序员可能不是我们通常想象的那样不善言辞。个人感觉，好的程序员一般逻辑思维能力很强，可能知识领域会跨域多个领域，比如浙江大学已故的陈教授天洲，心理学、计算机学都非常精通，即便患病期间还在国际顶级期刊上发表了3篇顶级的医学文章，陈教授安息!!!
- (4) 好的程序员通常有自己的私人的一些研究、爱好、项目，而这些大多数情况下他们不会写在简历上，但表现出来却可能恰恰是他的潜能、深度和后劲所在。
- (5) 技术多样性、先进性，在多种技术方向都有涉及，而且对多种技术的优劣有个人独到的见解，喜好尝试新鲜技术。云计算、大数据技术不就是这样产生的吗。
- (6) 基本上还是忽略证书吧，高手哪有时间去考证书，有这个闲工夫还不如多做点东西出来，多写点文章呢。

当然，以上6点可能有点片面，请读者自己多多体会，也欢迎微信讨论，本文作者微信号 michael_tec，欢迎添加。

8.3.2 如何成为技术大牛

当一个程序员的技术能力和解决问题的能力达到一定水平之后，他就能轻松胜任某些开发任务、解决特定实际问题，从而给用户带来某方面的便利。他的能力与他接触到的问题匹配，此时程序员处于工作满意度较高的状态，这个舒适区的大小是由他解决问题能力的大小定义和界定的。什么时候挑战的情况开始发生呢？当问题超出程序员先有技能和经验说明，程序员能看到并了解，但还不能解决，这个时候随时都可能给程序员带来挑战的感觉。这还只是挑战而已，如果我们引入了一个全新的编程语言，需要让程序员在有限时间内快速掌握该技能，那么这时才会进入真正的挑战区域，为了与前面的挑战区相区别，我们定义它为未知区域。对大多数程序员来讲，未知即痛苦。这个区域往往是程序员看不清或看不到的，是百慕大三角的一片未知而神秘的区域，贸然跳入，可能折戟沉沙铩羽而归。假如一个程序员愿意跳出舒适区，踏入挑战区，接受一定的不适，那他就能有机会拓展他的能力，将自己的舒适区扩展得更大，他的舒适区就会变大，挑战区也会跟着变大，未知区也会变大，这也是符合人类认知规律的：知道得越多，未知的也越多。如果一个程序员连轻微不适都不愿意接受，那他就会渐渐故步自封，落后于别人，落后于时代，渐渐被这个日新月异的时代所抛弃，成为一个别人眼中没什么用的老家伙。

一个程序员的能力，是可以通过锻炼不断变强的。就像人的肌肉，一段时间让锻炼强度超负荷一点，肌肉变得比原来强了，就再超负荷一点，通过这样的螺旋式递进，肌肉就会越来越强。程序员也是一样的，你的学习能力、代码能力、设计能力、沟通能力、管理能力等，都是可以通过锻炼来加强的（但我们也得考虑一个人适合做什么，如果他没有某方面的才干，虽然通过锻炼也可以加强，但违背天性的事儿通常为事倍功半）。在软件开发过程中，一个程序员，他会什么语言懂什么框架水平如何，自己心里有数，项目经理通过他的表现也认为自己心里有数。那么在新的项目要做时，通常的做法是，哪个程序员熟悉实现 Tx 任务相关的技术，就让这个程序员做 Tx 任务，这通常又是出于交付期、生产率、成本等各方面的考虑。在这种情况下，每个人都做自己驾轻就熟的事情，对整个项目来讲，自然是最经济的。可是对程序员自己来讲，却是不经济的。

因为你无法接受新的挑战，你的能力边界的拓展就会很慢。所以，合理的情况是，项目经理在划分任务时，要对程序员负责，既给一个程序员能轻松完成的任务，也要给他需要费点儿劲儿才能完成的任务，通过具有挑战性的任务来锻炼这个程序员，让他更好更快的成长。但是这样做的管理成本太高，所以，现实当中，很少公司的项目经理会主动这么做（没合适人手 **take** 某个任务时会被动这么做）。鉴于这种现实，作为程序员自己，如果你想更快地成长，就要表现得勇敢一些，主动走到挑战区域，去抢具有挑战性的任务。一旦你拿到了对你来讲具有挑战性的任务，虽然你会为此殚精竭虑，虽然你可能为此加班，虽然你可能为此在别人看不见的地方付出，但是你拥有了机会和更多可能性，如果你顺利完成了，那你的舒适区会扩大，你接触新挑战的机会也会变大，你就进入了良性循环，你会越来越强大。So，某个技术没搞过？不是问题。某个语言没学过？不是问题。软件结构太复杂，一时掌控不了？不是问题。业务不熟悉？不是问题。

在渴望成就自我的程序员眼里，问题即机会。只有抓住机会，我们解决问题的能力才会在痛苦的历练中像雪球一样越滚越大。

8.3.3 编程方法分享

编程是个实在活，千万不要以为买了一堆书，你就能自动成为大牛，你必须耐心地把这些书看完，不断练习、不断思考，这样你才有机会成为大牛。

对于初学者来说，我觉得基础知识非常重要，而这些基础知识不能光靠记忆，记忆是不可靠的，还是要学会自我思考，理解编程、设计的精髓。这就好像我们小时候背乘法口诀，老师天天追在屁股后面让我们背，光死背能背出来吗？我还记得读高中时候化学老师让我们把元素周期表全部背诵下来，差一点就磨灭了我的化学热情。

学习一门弱类型的编程语言，不要先学习那种具有强制类型的、面向对象的编程语言。严格而言，如果有人对你提到 **class**(类)或继承，那么你就应该去选择其他的途径了。虽然我认同类和继承相关技术是软件开发中必不可少的，但是我强烈认为它们不应该是初学者的选择。

考虑到这一点，这里有一些具体的建议给那些正在学习或准备学习 Web 应用开发的初学者。实际上，说得更远点更抽象点，这就是一个如何开始学习软件开发的一个好计划。很显然，这不是一个适合所有人的计划，但是我认为它一定适合大部分初学者。鉴于此，我认为 JavaScript 和 Java 是对于初学者而言最理想的编程语言，因为 JS 解释器在绝大部分浏览器上都可用，而 Java 只需要安装 JDK 和 Eclipse 这样的 IDE 工具就可以直接调试，再则 JavaScript 的面向对象特性并不是强制型的，并且无论 Java 或 JavaScript，它们都在工业界被广泛使用。

以 JavaScript 为例，说得更具体点，我建议你以这个顺序学习。

学习如何打印出一些东西，学习如何声明和定义变量，学习基本算术运算操作（包括余数操作），学习循环（特别是 for 循环），学习把抽象重复的代码写成函数，学习字符串和用循环操作字符串，学习数组和数组的循环方法（特别是 **foreach** 循环），学习创建和操作对象数据集。记住上面的这些并每天写一个程序来实践，直到这些都轻而易举地想起来。学习 Git 的基本操作，学习通过命令行使用 Git。这意味着要先学习四个 Unix/Linux 命令（**ls,pwd,mkdir,cd**）。当学习了这几个命令，也就学会了以“树型”或层次结构的呈现方式查询文件系统。

一旦你掌握了上面的几个 Unix/Linux 命令，并会从命令行进入文件系统，你就应该学几个基础的 Git 命令。主要是 **git init, git status, git add and git commit**。一旦你掌握了 Git 的基本操作，在

学习下面的技术时将其集成到你的工作流中。学习 HTML 基础，能够凭记忆创建简单的 HTML 页面。学习 DOM 和如何理解 HTML 作为指定的分层树结构。花点时间来思考它如何关系到你在前面步骤中学到的分层文件系统。学习 CSS 选择器，了解它如何让你选定 DOM 的某些部分。了解 DOM 元素之间的关系。了解一个 DOM 元素作为另一个 DOM 元素的父元素或子元素的含义。理解这与后代和祖先之间的关系有什么不同。记住选择器可以让你通过这些关系来选定某些元素。学习 jQuery，并主要专注于 DOM 的操作能力。学会用 jQuery 对 DOM 插入或删除元素，实践可视化如何影响用 DOM 定义的树型结构。实践 jQuery 中的事件处理和 DOM 操作（比如，实践操作 DOM 当用户点击某个东西，或在指定的时间间隔）。多练习 JavaScript 对象，并把它们当作可变的聚合器。学习如何用 JavaScript 来表示更复杂的数据而不是基本数据类型。学会应用并操作这些数据结构。理解并定义 JSON、理解它如何与 JavaScript 对象相关联。学会使用 jQuery 的 getJSON 函数从文件中获取数据到 JavaScript 对象中。使用类似的技术，用一个简单的 JSONP API 去练习用 AJAX 拉取数据。练习向 DOM 插入和删除这个数据。在这个阶段，做一个简单的幻灯片来循环播放 Flickr 图片，这将是一个令人难以置信的项目，将真正考验你的能力，使用之前学过的基础技术来实现它。如果你做了这一步，那么你已经掌握了大量必备的编程和计算机科学基本概念。具体来说，你掌握了计算机程序的最重要元素（如果 if-else 语句，循环，变量，对象，函数，数组等），你已经学会了链式或树型的数据结构。这时，无疑你已经准备好转移到更高级的主题。

8.4 本章小结

本章是针对前面各个章节没有提到的一些性能优化建议、全局性建议的补充。首先针对 Web 应用、Web 容器的性能优化提出自己的建议，然后讲解了一些数据库应用方面的优化建议，接下来对企业级应用和系统整体架构方面的优化提出自己的看法，最后是一些与个人品质、思维方式相关的建议。通过本章的分享，全书完成了所有与 Java 程序相关的知识分享，希望读者能够受益。

十载耕耘奠定专业地位

博文视点诚邀精锐作者加盟

以书为证彰显卓越品质

《C++Primer (中文版) (第5版)》、《淘宝技术这十年》、《代码大全》、《Windows内核情景分析》、《加密与解密》、《编程之美》、《VC++深入详解》、《SEO实战密码》、《PPT演义》……

“圣经”级图书光耀夺目,被无数读者朋友奉为案头手册传世经典。

潘爱民、毛德操、张亚勤、张宏江、咎辉Zac、李刚、曹江华……

“明星”级作者济济一堂,他们的名字熠熠生辉,与IT业的蓬勃发展紧密相连。

十年的开拓、探索和励精图治,成就博古通今、文圆质方、视角独特、点石成金的计算机图书的风向标杆:博文视点。

“凤翱翔于千仞兮,非梧不栖”,博文视点欢迎更多才华横溢、锐意创新的作者朋友加盟,与大师并列于IT专业出版之巔。

英雄帖

江湖风云起,代有才人出。

IT界群雄并起,逐鹿中原。

博文视点诚邀天下技术英豪加入,

指点江山,激扬文字

传播信息技术,分享IT心得

• 专业的作者服务 •

博文视点自成立以来一直专注于IT专业技术图书的出版,拥有丰富的与技术图书作者合作的经验,并参照IT技术图书的特点,打造了一支高效运转、富有服务意识的编辑出版团队。我们始终坚持:

善待作者——我们会把出版流程整理得清晰简明,为作者提供优厚的稿酬服务,解除作者的顾虑,安心写作,展现出最好的作品。

尊重作者——我们尊重每一位作者的技术实力和生活习惯,并会参照作者实际的工作、生活节奏,量身制定写作计划,确保合作顺利进行。

提升作者——我们打造精品图书,更要打造知名作者。博文视点致力于通过图书提升作者的个人品牌和技术影响力,为作者的事业开拓带来更多的机会。



联系我们

博文视点官网: <http://www.broadview.com.cn>

CSDN官方博客: <http://blog.csdn.net/broadview2006/>

投稿电话: 010-51260888 88254368

投稿邮箱: jsj@phei.com.cn



新浪微博
weibo.com

@博文视点Broadview



微信公众账号 博文视点Broadview



博文视点精品图书展台

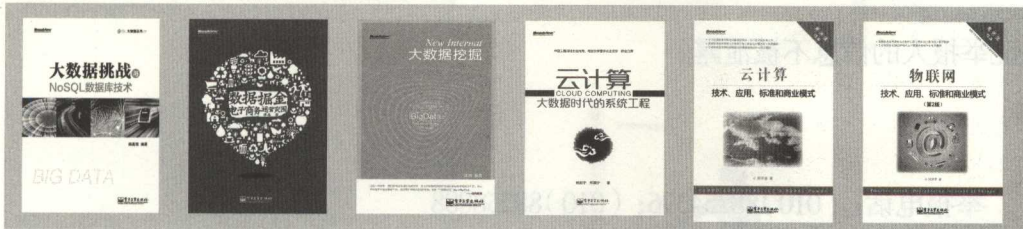
专业典藏



移动开发



大数据 · 云计算 · 物联网



数据库



Web 开发



程序设计



软件工程



办公精品



网络营销



反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序,保护权利人的合法权益,我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为,本社将奖励举报有功人员,并保证举报人的信息不被泄露。

举报电话: (010)88254396; (010)88258888

传 真: (010)88254397

E-mail: dbqq@phei.com.cn

通信地址:北京市万寿路173信箱 电子工业出版社总编办公室

十 十 邮 编: 100036 十 十 十 十

好书分享



欢迎反馈意见或投稿
邮箱: dongying@phei.com.cn
电话: 010-88254047
微信号: yingzidd

专家热评

系统调优在软件的后续改进和重构中占有很重要的地位，能够弥补软件的不足，本书以通俗的语言和引人入胜的故事，重点讲述软件性能调优的方法论和具体实现路径，读者可以根据自己的实际情况进行参照比对，就像进了兵器库挑选适合自己的顺手武器。

程序凑合着上线是一回事，而在压力下能够优美地运行往往很不容易。本书对于所有有志于进行软件高级管理的人员而言，具有非常重要的意义。

海适云承CEO兼首席架构师 沈英桓 (Sam Shen)

当我翻开周明耀先生编写的《大话Java性能优化》这本书时，一下子被他生动朴实的语言所深深吸引，他将生硬、深奥的IT系统技术问题深入浅出地层层剥开，娓娓道来，并结合时下大家最为熟知的12306、电商等案例，系统地分析和介绍了系统调优的重要性、解决思路和技术实现。

作为金融IT的一名同行，我对系统性能对用户体验和业务处理的重要性深有体会，尤其是高频交易系统（HFT），对系统性能的要求近乎苛刻，对业务的处理和响应要求毫秒级。本书作者从系统架构、系统设计、开发、编码、算法等多层次、多角度提供思路和优化策略，是一本很务实的技术贴，值得大家学习、借鉴和探讨。

德意志银行（中国）有限公司环球科技运营经理 黄正兵

在我自己使用Java开发项目的过程中，经常会切实地感受到系统调优的重要性。然而Java性能调优并不是一项一蹴而就的简单任务，而是如同并发编程需要关注算法、内存、I/O等各种问题，以及丰富的经验积累。

本书中作者结合自己的实践经验总结了一些性能优化的方案。这些经验涉及Java基本语法、对象和引用、String类型和集合类的使用等各个方面且附有示例，使人受益匪浅，如果能够将其灵活运用到自己的系统中，相信能够对读者处理性能优化问题提供不小的帮助。此外，作者看待性能优化问题的视角相对开阔，系统且详尽地讨论了可能导致性能问题的各个环节和不同角度下性能优化的问题，读后令人豁然开朗。

西安工业大学2016应届硕士毕业生 Fenny



博文视点Broadview



@博文视点Broadview



责任编辑：董英
封面设计：李玲

上架建议：计算机>Java

ISBN 978-7-121-28481-6



9 787121 284816 >

定价：89.00元